PolyAPM: Parallel Programming via Stepwise Refinement with Abstract Parallel Machines

Nils Ellmenreich and Christian Lengauer
{nils,lengauer}@fmi.uni-passau.de

Fakultät für Mathematik und Informatik, Universität Passau, Germany

Abstract. Writing a parallel program can be a difficult task which has to meet several, sometimes conflicting goals. While the manual approach is time-consuming and error-prone, the use of compilers reduces the programmer's control and often does not lead to an optimal result. With our approach, PolyAPM, the programming process is structured as a series of source-to-source transformations. Each intermediate result is a program for an Abstract Parallel Machine (APM) on which it can be executed to evaluate the transformation. We propose a decision tree of programs and corresponding APMs that help to explore alternative design decisions. Our approach stratifies the effects of individual, self-contained transformations and enables their evaluation during the parallelisation process.

1 Introduction

The task of writing a program suitable for parallel execution consists of several phases: identifying parallel behaviour in the algorithm, implementing the algorithm in a language that supports parallel execution and finally testing, debugging and optimising the parallel program. As this process is often lengthy, tedious and error-prone, languages have been developed that support high-level parallel directives and rely on dedicated compilers to do the low-level work correctly. Taking this approach to an extreme, one may refrain entirely from specifying any parallelism and use instead a parallelising compiler on a sequential program. The price for this ease of programming is a lack of control over the parallelisation process and, as a result, possibly code that is less optimised than its hand-crafted equivalent.

We can view the process of writing a parallel program as a sequence of phases, many of which have alternatives. Selecting a phase from among several alternatives and adjusting it is called a *design decision.*

Both of these opposing approaches – going through the whole parallelisation process manually or leaving the parallelisation to the compiler – are unsatisfactory with respect to the design decisions. Either the programmer has to deal with the entire complexity, which might be too big for a good solution to be found, or one delegates part or all of the process to a compiler which has comparatively little information to base decisions on.

Even if a parallelising compiler honours user preferences to guide the compilation process, it is difficult to identify the effect of every single option on the final program. There is no feasible way of looking into the internals of a compiler and determining the effect of a design decision on the program.

To avoid the aforementioned drawbacks of current parallel program development paradigms, we propose an approach that combines the advantages of manual and automatic parallelisation. To bridge the gap between the algorithm and the final parallel program, we introduce a sequence of *abstract machines* that get more specific and closer to the target machine architecture as the compilation progresses. During the compilation, the program is undergoing a sequence of source-to-source transformations, each of which stands for a particular compilation phase and results in a program designed for a particular APM. This makes the compilation process more transparent since intermediate results become observable by the programmer. These observations may influence the design decisions the programmer makes as the compilation progresses further. We have implemented simulators for the machines, thus enabling the programmer to evaluate the result of each transformation directly on the executing program. Our experience has been that this result structures parallel program development and that it helps evaluating the effects of a single decision during the parallelisation process.

This paper is organised as follows: Section 2 gives a brief description of the use of APMs in program development. Our PolyAPM approach and the APMs we have designed are presented in Section 3. An example of a parallelisation by using the APMs is given in Section 4. Section 5 describes the experiences we made using APMs. An overview of related work is given in Section 6. The paper is concluded by Section 7.

2 Program Development using Abstract Machines

The idea of stepwise refinement of specifications has long been prevalent in computer science. Trees are used to represent design alternatives. If we look at the various intermediate specifications that exist between all the transformations of the parallelisation process, we observe an increasing degree of concreteness while we proceed. Thus, the abstract specification is finally transformed into a binary for a target machine. Consider the intermediate steps: on our descent along one path down the tree, we pick up more and more properties of the target architecture. But that also means that, most likely, no existing machine matches the level of abstraction of any intermediate specification. If we employ an abstract machine model that is just concrete enough to cover all the details of our program, we can have it implemented in software and even run our intermediate programs on it.

One specific kind of abstract machine has been described by O'Donnell and Rünger in [OR97] as the *Abstract Parallel Machine* (APM). They define it by way of the functional input/output behaviour of a *parallel operation* (ParOp). Their notion of an APM is closely related to its implementation in the functional language Haskell. The use of Haskell is motivated by its mechanisms for dealing with high-level constructs and by a clearly defined semantics that make proofs of program transformations feasible. However, we would like to model machine characteristics more closely within the APM and will base our work only loosely on [OR97].

3 PolyAPM

Rather than using just one APM in the sense of [OR97] for all transformations, we need to design variations of APMs. The compilation process is a sequence of source-to-source transformations each of which describes one particular step in the generation of a parallel program. Therefore, we need levels of abstraction corresponding to the machines properties assumed by the program transformations. The sequence of programs is associated with a sequence of APMs. In general, there are fewer APMs than programs, since no every transformation introduces new machine requirements.

3.1 The PolyAPM Decision Graph

As discussed above, a single problem specification may lead to a set of possible target programs, mainly because different parallelisation techniques and parameters are used and different target architectures are to be met. Therefore, the process of deriving a target program is like traversing the tree of design decisions. However, in certain cases, two different branches may lead to the same program, thus making this tree a DAG, the PolyAPM Decision Graph (PDG). Each node in this graph is a transformed program that runs on a dedicated APM. There are two graphs: one for the APMs themselves and one for the APM programs, where each node in the former may correspond to several nodes in the latter. A part of the PDG is given in Figure 1. The transformations depicted in the PDG have been motivated by our experiences with the *polytope model* for parallelisation [Len93] within the LooPo project [GL96], but PolyAPM is not restricted to this model.

The program development process is divided into several phases as follows:

1. Implementation of a problem specification in standard sequential Haskell as a *source program*. There are two main reasons for using Haskell. First, we claim, that for many algorithms that are subject to parallelisation (first of all numerical computations), the Haskell implementation represents a natural "rephrasing" of the problem in just a slightly different language. Second, as we use Haskell to implement the APMs, the APM program's core also is a Haskell function that is being called by the APM interpreter. It should be avoided to cross another language barrier in order to obtain the first APM program. However, these reasons make the choice of Haskell only highly suggestive, but not necessary.

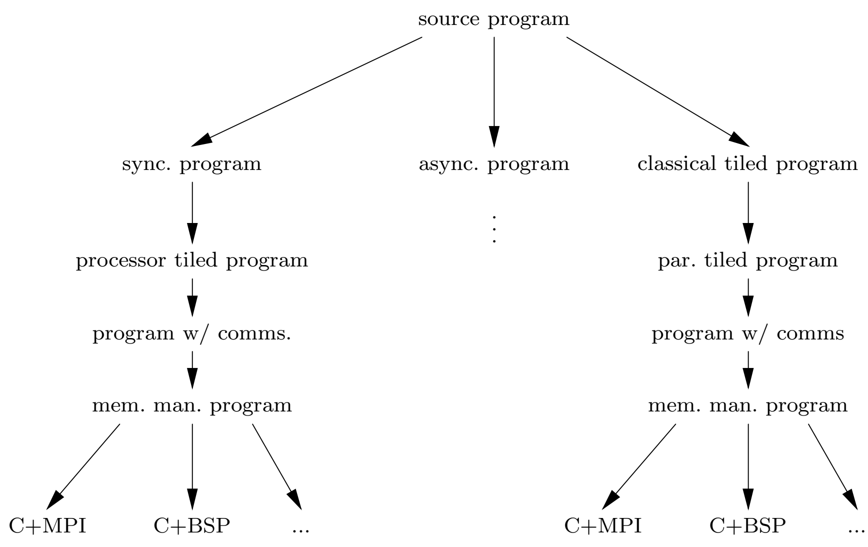

Fig. 1. PolyAPM Decision Graph (PDG), here only a sub-tree

2. Initial parallelisation of the sequential program. It means the analysis of the problem to identify independent computations that can be computed in parallel. This process might be done manually (as is the case with our example in Section 4), or with help of a parallelisation tool. We have used LooPo [GL96] for this purpose. In any case, the result of the parallelisation should map each computation to a virtual processor (this mapping is called *allocation*) and to a logical point in time (*schedule*). The granularity of the computation is the choice of the programmer, as the PolyAPM framework will maintain this granularity throughout the process. As the source program will most likely contain recursion or a comprehension, it is often sensible to perform the parallelisation on these and keep the inner computations of the recursion/comprehension *atomic*.
   Without loss of generality, we assume a one-dimensional processor field so that we have the basic computations, their allocation in *space* (i.e., the processor) and their scheduled computation time. With these components, the problem has a natural expression as a loop program with two loops, where the processor loop is parallel, the time loop is sequential and the loop body is just our atomic computation. If the outer loop is sequential, we call the program *synchronous*, otherwise *asynchronous*. This motivates the corresponding branches of the PDG in Figure 1. The right branch for classical tiling is a special case where parallelisation is not the first step.
3. Based on the parallelisation, the source program is transformed into an APM program, which resembles an imperative loop nest with at least two surrounding loops (there may be additional loops in source program's core computation). The program is subject to several transformations to adapt

$$\begin{tikzcd}
 & \text{source program} \arrow[dl, dashed] \arrow[dr, dashed] & \\
\text{SynAPM} \arrow[d] & & \text{AsynAPM} \arrow[d] \\
\text{SynCommAPM} \arrow[d] & & \text{AsynCommAPM} \arrow[d] \\
\text{SynMMAPM} & & \text{AsynMMAPM}
\end{tikzcd}$$

Fig. 2. PolyAPM Machine Tree

it to other APMs. This is the central part of PolyAPM and is discussed in more detail below in Section 3.2.

4. The final result of the compilation, the *target program*, has to be executable on a parallel machine. Therefore the last APM program is transformed into a *target language* for the parallel machine. It is important that the target language exhibits at least as much control as the last APM needs, so that no optimisation of any APM program transformation is lost. Suitable target languages, among others, are C+MPI and C+BSP.

3.2 Abstract Machines and their Programs

The APMs form a tree, as shown in Figure 2. There is a many-to-one mapping from the programs to APMs. An APM program must reflect the design characteristics of the corresponding APM, e.g., in case of a synchronous program, a loop nest with an outer sequential and an inner parallel loop and a loop body, which may contain more loops. This separates the loops representing the real machine from logical loops.

The synchronous program is subject to a sequence of source-to-source program transformations. Each adds another machine characteristic or optimises a feature needed for execution on a real parallel machine. Assuming that the original parallelisation was done for a number p of processors whose value depends on the input, the p processor's workload has to be distributed on rp real processors of the target machine. This transformation is called *processor tiling*, in contrast to tiling techniques with other purposes.

The next two transformations complete the transition to a distributed memory architecture with communications. This has been deliberately split up into two transformations: First, while still maintaining a shared memory, we'll be generating communication directives. As a second step, the memory is distributed, making the communication necessary for a correct result. The reason for this unusual separation is twofold: one of the aims of PolyAPM is to make each transformation as simple as possible, and both, communication generation and memory distribution, can get complicated. Furthermore, if we did both transformations in one step and the resulting APM program had an error, it would be more difficult than necessary to isolate the reason of this error. When interpreting an APM program that communicates even in the presence of shared memory, the communications perform identity operations on the shared memory cells. The APM interpreter checks for this identity and issues a warning in case of a mismatch. This way wrong communications will be detected, but missing communications show their effects only after the distribution of memory.

The transformed program will have to run on an APM capable of communications, the SynCommAPM, which provides a message queue and a message delivery system. We assume that each processor stores data in local memory by the *owner-computes rule*. Therefore, data items computed elsewhere have to be communicated, either by point-to-point communications or by collective operations. If we were to employ a different storage management rule, this transformation would have to be adapted accordingly.

The "unnecessary" communications of the SynCommAPM program become crucial when the memory is being distributed for the SynMMAPM-program. This branch of the tree uses the owner-computes rule, making it easy to determine which parts of the global data space are actually necessary to keep in local memory. That completes the minimal necessary set of transformations needed for a synchronous loop program on a distributed-memory machine. The last transformation generates so-called *target code*, i.e., it transforms the SynMMAPM program into non-APM source code that is compilable on the target machine. Possible alternatives include C+MPI and C+BSP.

As outlined in Figure 1, transformation sequences other than the synchronous one are possible. In addition to the corresponding asynchronous one, we have depicted a typical sequence as employed by the tiling community [Wol87].

4 Example/Case Study

As an illustrating example, we chose the one-dimensional finite difference method. We start with an abstract problem specification, and by going through the process of program transformations – each yield a new, interpretable specification – we will eventually obtain an executable program for a specific target platform.

First we need to implement the specification in Haskell and identify the parallelism. The APM program expresses the parallelism as a loop nest with one of the two outermost loops being tagged as "parallel". Then we derive subsequent APM programs until a final transformation to the target language is feasible.

The abstract specification of the one-dimensional finite difference problem (as presented by [Fos95]) describes an iterative process of computing new array elements as a combination of the neighbour values and the previous value at the same location. Formally, the new array a_t with p elements at iteration t is defined as:

$$(\forall i \in \{2 \dots p-1\} :: a_t[i] := (a_{t-1}[i-1] + 2 * a_{t-1}[i] + a_{t-1}[i+1])/4) \qquad (1)$$

```
findiff:: Array Int Float -> Array Int Float
findiff a = listArray (low,up)
            ( a!(low):
             [(a!(i-1)+2*a!(i)+a!(i+1))/4 | i<-[low+1..up-1]] ++[a!(up)])
    where (low,up) = bounds a

findiffn:: Int -> Array Int Float
findiffn n  = f n
    where f:: Int ->Array Int Float
          f 0 = findiff testinput
          f n = findiff (f (n-1))
```

Fig. 3. Sequential Haskell Specification of Finite Differences

```
loop_syn = LP [(Seq, \([],[n])->0, \([],[n])->20, 1),
               (Par, \([t],[n])->0, \([t],[n])->n, 1)]
              (BD body_syn)

body_syn::((Array Int Float,Array Int Float),[Idx]) -> [Idx]
       -> ((Array Int Float,Array Int Float),[Idx])
body_syn  ((a,a1),splist) [t,p]
         | (p == low)||(p == up) = ((a,a1//[(p,a!p)]),splist)
         | (low < p )&&(p <  up) = ((a, stmnt1 a a1 (t,p,n)),splist)
    where stmnt1 a a1 (t, p, n)  = a1 //[(p, (a!(p-1)+2*a!(p)+a!(p+1))/4)]
          (low,up) = bounds a
          [n] = splist
```

Fig. 4. SynAPM version of the Finite Differences

The Haskell specification of this problem is given in Figure 3. Note how closely the Haskell program of Figure 3 corresponds to Equation 1. `findiff` represents one iterative step in which a new array is computed from the previous array a, just as Equation 1 requires. Function `findiffn` calls `findiff` as often as the parameter n prescribes. Each call to `findiff` generates a new Haskell array with the updated values. This corresponds to a destructive update of an array using an imperative programming model. In this example, all references to a refer to the previous iteration, so that we need to copies of the array.

We continue with the presentation of transformed `findiff` programs, emphasising the difference to the respective predecessor programs.

4.1 The Synchronous Program

The parallelisation of the source program is done manually. It is obvious that the calculations of the array elements of one particular `findiffn` call are independent, but they all depend on the previous values. Thus, `findiffn` corresponds to

```
body_til::((Array Int Float,Array Int Float),[Idx]) -> [Idx]
        -> ((Array Int Float,Array Int Float),[Idx])
body_til ((a,a1),splist) [t,p,p2]
    | (p2==low)||(p2==up) =  ((a,a1//[(p2,a!p2)]),splist)
    | (low< p2)&&(p2< up) =  ((a, stmnt1 a a1 (t,p2,n) ),splist)
    where stmnt1 a a1 (t, p, n) =  (a1 //[(p, (a!(p-1)+2*a!(p)+a!(p+1))/4)])
          (low,up) = bounds a
          [n] = splist

loop_til = LP [(Seq, \([],[n])->0, \([],[n])->20, 1),
               (Par, \([t],[n])->0, \([t],[n])->physprocs, 1),
               (Seq, \([t,p],[n])->max (p*(tilesize_p n)) 0,
                     \([t,p],[n])->min ((p+1)*(tilesize_p n)) n, 1)]
              (BD body_til)
                   where
                   tilesize_p:: Int -> Int
                   tilesize_p n = min ( n `div` physprocs) n
```

Fig. 5. Tiled SynAPM version of the Finite Differences

an outer, sequential loop, whereas the list comprehension inside `findiff` yields a parallel loop.

To write a SynAPM program for `findiff`, we proceed as follows:

1. We define the memory contents, here: two arrays `a` and `a1` of the same kind as the input array,
2. We set the read-only structure parameter list to n, which describes the size of `a`,
3. We define the loops: one outer sequential loop, arbitrarily set to 21 iterations, and one inner parallel loop, ranging from 0 to $n-1$, and
4. We write a loop body function.

This defines the synchronous loop nest `loop_syn` in Figure 4. The body function of SynAPM has the following type: `BD (e -> b -> e)`, i.e., it takes some *state*, consisting of memory and structure parameters, and a list of current values of all surrounding loops, to return an updated state. Figure 4 shows the state to be of type `(Array Int Float,Array Int Float),[Idx])`. The first array `a` is the one computed by the last iteration, and `a1` is filled at this time step. Note that the structure parameter list `splist` consists of only one item: the size `n` of the array. As the computation takes place only on the inner array values, we need a case analysis to take care of the border cells.

4.2 The Tiled Program

Figure 5 shows the synchronous `findiff` program after a simple processor tiling transformation. The parallel loop has been partitioned into tiles such that the

```
body_comm = similar to body_til
loop_comm = similar to loop_til, msg generation after each body

instance Sendable SC_Dom Int Float SCMem where
  generateMsg [t,p,p2] [n] (a,a1) =
              [Msg (p2, to_p, t, to_tm, A, p2, a1!p2)|
                to_p <- (case pos_in_tile of
                            pat | pat==low    -> []
                                | pat==up     -> []
                                | pat==0      -> [p-1]
                                | pat==(ts-1) -> [p+1]
                                | True        -> [] ),
                to_tm <- [t+1]
              ]
      where (low,up)= bounds a
            pos_in_tile = p2 `mod` ts
            ts = tilesize_comm n

instance Updatable SC_Dom Int Float SCMem where
  updateMem (Msg (from_p, to_p, from_tm, to_tm, dom, idx, val)) (a,a1) =
         if (a1!idx) == val then (a,a1//[(idx,val)])
                            else wrong update
```

Fig. 6. SynCommAPM version of the Finite Differences

number of remaining parallel iterations matches the number of physically available processors (as defined by `physprocs`). An additional sequential inner loop $p2$ has been added that enumerates the previously parallel iterations sequentially. Its bounds make sure that the loop variable takes the values previously provided by loop p, so that the relevant parameters of the body now are t and $p2$. Other than that, the function `body_til` is identical to `body_syn`. As the APMs work with an arbitrary but fixed number of processors, the tiled program can still run on SynAPM.

Changes to the code to obtain the *tiled program*:

- The constant `physprocs` (denoting the number of physical processors on real machine) and the `tilesize` function are added.
- An additional loop in `LP` with loop variable $p2$ is added.
- The body function takes three loop variables, p is replaced by $p2$.

4.3 The Communicating Program

Loop and body functions are the same as in the tiled program except for the memory type. Whereas in the previous APM programs the memory type could be freely defined, we now require the parameterised type `State` that couples memory and message queue. The parameters are required by context declarations within the SynCommAPM interpreter. See Figure 6. Emphasised font is used for pseudo code that replaces some longer Haskell code. It is meant to shorten the presentation.

New are three additional functions that have to be implemented by the programmer and which are called from inside the interpreter:

– `generateMsg` generates new messages originating from each processor at each time-step;
– `updateMem`, updates the state's memory with values sent by a message;
– `synchronizeMem`, being called after each time-step for possible synchronisation work on all processors. In this case, `synchronizeMem` removes the old array in each processor's state and introduces an empty new one to be filled in by computations in the next time-step. `synchronizeMem` will be omitted in the given code examples as it is just an auxiliary function.

These three functions have to be introduced by class instance declarations because the APM interpreter needs some type information to use the – at this point – undefined functions as stubs. This is because the APM interpreters reside in a separate Haskell module that is being used by different APM program modules. So, for every APM program the specific instances of these three functions are different, yet they need to fit into the APM, and making them instances of multi-parameter type classes guarantees the integration into the APM interpreter.

Function `updateMem` checks before an update whether the memory's and the message's values are identical. If they are not, the interpreter issues a runtime error message because a wrong communication message was generated. This is no method to prove correctness of communications, but testing the SynCommAPM program with a variety of inputs without errors can provide some confidence in the message generation, which belongs to the more error-prone parts of parallel programming. The SynMMAPM will provide further communication checks.

Changes to the code to obtain the *communicating program*:

– A `State` type combining memory and message queue replaces the memory; types in body and `LP` are adapted accordingly.
– Each call of the body is followed by a call of the message generation function of SynCommAPM, which in turn calls the provided `generateMsg`.
– Instance declarations for `generateMsg`, `updateMem` and `synchronizeMem` are added.

4.4 The Memory-Managed Program

With the paradigm shift from shared to distributed memory, the memory representation in the APM programs has to be adapted. Each processor gets its own chunk of the memory, which in the example in Figure 7 comprises all the data which is computed on this processor and the remotely-owned data that

```
data MM_PProcMem = PPM (Array Int Float)  (Array Int Float)

body_mm::([Idx],[Idx]) -> MM_PProcState ->  (MM_PProcState,[Idx],[Idx])
body_mm (splist, idxlist@[t,p,p2]) (State (PPM a1 a2) msgs)
    | (p2==low)||(p2==up) =
         (State (PPM a1 (a2//[(lidx,a1!lidx)])) msgs,splist,idxlist)
    | (low <p2)&&(p2 <up) =
         (State (PPM a1 (a2//[(lidx,stmnt1)])) msgs,splist,idxlist)
    where stmnt1 = (a1!(lidx-1) + 2*a1!lidx + a1!(lidx+1))/4
          (low,up) =  (0,n-1)
          [n] =  splist
          lidx = (proc2idx p2 n)

The functions loop_mm, generateMsg and updateMem are similar
to before and have only been adjusted to the new memory data type.
```

Fig. 7. SynMMAPM version of the Finite Differences

is required for the computation. The values of the latter will be communicated before the computation. So the one-dimensional array is divided into chunks according to the tile size, with an additional element to the left an to the right, because each element's computation needs its two neighbours. The body function's indexing into its local array has to be adapted. The index range changed from $\{1 \ldots n-1\}$ to $\{1 \ldots \mathit{tilesize}-1\}$. Furthermore, all communications within a tile can be eliminated, thus requiring a change in the `generateMsg` function. This program resembles very much an imperative SPMD program with loops as control structure so that the transition to C+MPI is relatively straightforward.

Changes to the code to obtain the *memory-managed program*:

- The memory type within the global state changes to an array of local memory types. Body, `generateMsg` and `updateMem` are changed accordingly.
- In `LP`, just the names of the body/generateMsg functions changes.

4.5 The C+MPI program

This last transformation leaves the APM realm. Conceptually, nothing interesting happens, but a language barrier has to be crossed. The simpler the body function is, the easier its transformation into a C function gets. The premier area of parallel programming, scientific computation, usually deals with arithmetic operations on arrays. The array as the most frequently used data structure exists in both languages. This is not to say that more general problem domains cannot be handled, but then the target code transformation gets more complicated.

To generate target code, abstract APM communications have to be transformed into MPI calls, the memory data type and its distribution/aggregation function need the imperative equivalent. But all these changes are isolated, and in most cases not difficult, especially if this transformation was taken into account while choosing the appropriate Haskell types.

Changes to the code to obtain the *C+MPI program*:

- A template for an SPMD program in C+MPI for the APM structures is provided.
- The memory type has to be adapted, the message queue is subsumed by MPI.
- The functions `generateMsg`, `updateMem` and `synchronizeMem` are rewritten in C and replace the stubs within the C template.

5 Critical Evaluation

In Section 4 we showed a simple development in a straight sequence of programs. In practise, one will want a choice between sequences, leading to a tree as suggested. Still, here is a preliminary evaluation of the PolyAPM approach, based on our sequence example.

5.1 Benefits

- Effects of program transformations can be isolated and evaluated, to help deciding for the most suitable transformation path. An example is the communication generation for SynCommAPM. For complex programs, different communication patterns can be tested by executing the different versions of the communicating program. The APM interpreter can output communication statistics to help selecting the most suitable pattern with the smallest total number of communications.
- Building a test environment for program transformations becomes easier, as input and output of each transformation can be executed and compared with other transformations. PolyAPM can be used to construct a compilation system in which not all transformations are automatic, allowing incremental development. Alternatively, the structure of PolyAPM supports parallel compiler research where a complete compilation is not feasible and some transformations are performed manually, which is the case for all transformations in Section 4.
- Using Haskell has the benefit that the definition of the APM programming language as an algebraic data type provides syntax and type checks for free. Because of this, the APM programs and their interpreters can be kept relatively small, as no parsing of APM programs is necessary.
- It is a challenge to split up the Haskell program into the APM interpreter module and APM program modules. The interpreter has to make assumptions about the unknown APM program (especially that the three user-defined functions introduced by SynCommAPM exist and have types of a certain kind). Haskell's multi-parameter type classes support these assertions.

– A researcher who wants to prove that a transformation of APM programs preserves correctness, can do so with equational reasoning techniques.

5.2 Drawbacks

– It is a lot of effort to write all APM programs (for example four plus one sequential program in Section 4) in order to get one target program if the aim is just a single compilation. PolyAPM is not meant for the development of individual application programs.
– In PolyAPM the loop body has to be a Haskell function to maintain the generality of the approach (i.e., if we devise a special language for array assignments, which could be more easily transformed than a general Haskell function, then we severely restrict the class of applicable problems). The more non-trivial and Haskell-specific code the body function contains, the more problems arise when this code is transferred to an imperative language (see Section 4.5).

5.3 Who may profit from PolyAPM?

– Researchers who are interested in comparing the effects of transformations. This could be compiler writers or researchers in compilation and parallelisation techniques.
– Programmers who have a PolyAPM compilation systems at their disposal which can perform some transformations automatically. Programming the remaining (if any) transformations manually might be less work than writing the target program directly.
– Programmers who need to compile one source program for different target languages or different machine architectures and who have at least some transformations automated.

6 Related Work

John O'Donnell and Gudula Rünger presented APMs in [OR97] and provided a starting point for others to work on parallel compilation using these.

Joy Goodman has extended the above work [Goo01], included input and output via monads, investigated the decision-making process and formalised the decision making process.

Noel Winstanley also uses the APM methodology in his PEDL system [Win01]. He compiles array-based numerical programs to the parallel, imperative target language SAC. However, he uses a special restricted language, tailored for his specific problem domain, and focuses on a high degree of optimisation and automation of the compilation.

Many automatic parallel compilation systems exist with varying degrees of user interaction. Some research systems like SUIF[WFW$^+$94] serve as a compiler's workbench, where some compilation phases may be replaced by own implementations. The parallelizer LooPo[GL96] developed in our group also belongs to this category. Other systems like Polaris[BEF+95], Parafrase[PGH+90] and HPF compilers like Adaptor[Bra98] employ a more static view on the parallelisation, in which the selection of transformations is rather fixed.

7 Conclusions

Based on our initial experimental evaluation, we envision the PolyAPM model for specific sub-areas of of parallel program development rather than claiming a general purpose approach.

Structured (parallel) program development: The task of obtaining parallel target code consists of several steps. Current commercial parallelising compilation systems (mainly for HPF) often are only able to perform the compilation in one pass. There is usually no or only restricted influence on the selection of the used algorithms. This static process can be made flexible with a modular system of compilation phases, where single phases can be chosen from a given set of alternatives. A few academic systems use this approach for some phases of their compilation system.

Exploration of design decisions: Each phase in the PolyAPM compilation process is a source-to-source transformation on APM programs. These programs are executable by the respective APM machine interpreter. As a result, the effects of each transformation can be observed directly by looking at the code and executing it.

Rapid prototyping: Researchers with a specialised focus on only one phase of parallel program development can choose an APM at the abstraction level they need and evaluate their work without the need to write a full compiler.

Although, in this paper, program development has only been demonstrated for a single branch of the PDG, the full power of the approach is obtained by making use of a subtree or sub-DAG of the PDG.

8 Acknowledgements

We thank Paul Feautrier and John O'Donnell for fruitful discussions, the DFG for funding PolyAPMand PROCOPE for supporting the contact with Feautrier.

References

[BEF+95] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. Polaris: Improving the effectiveness of parallelizing compilers. In Keshav Pingali, Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, LNCS 892, pages 141–154. Springer-Verlag, 1995.

[Bra98] Thomas Brandes. *ADAPTOR Programmer's Guide, Version 6.0*, June 1998. Available via anonymous ftp from ftp.gmd.de as gmd/adaptor/docs/pguide.ps.

[Fos95] Ian Foster. *Design and Building Parallel Programs.* Addison-Wesley, 1995.

[GL96] Martin Griebl and Christian Lengauer. The loop parallelizer LooPo. In Michael Gerndt, editor, *Proc. Sixth Workshop on Compilers for Parallel Computers (CPC'96)*, Konferenzen des Forschungszentrums Jülich 21, pages 311–320. Forschungszentrum Jülich, 1996.

[Goo01] Joy Goodman. *Incremental Program Transformations using Abstract Parallel Machines.* PhD thesis, Department of Computing Science, University of Glasgow, September 2001.

[Len93] Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[OR97] John O'Donnell and Gudula Rünger. A methodology for deriving abstract parallel programs with a family of parallel abstract machines. In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *EuroPar'97: Parallel Processing*, LNCS 1300, pages 662–669. Springer-Verlag, 1997.

[PGH$^+$90] Constantine Polychronopoulos, Milind B. Girkar, Mohammad R. Haghighat, Chia L. Lee, Bruce P. Leung, and Dale A. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran. In David Gelernter, Alex Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'90)*, Research Monographs in Parallel and Distributed Computing, pages 423–453. Pitman, 1990.

[WFW$^+$94] Robert P. Wilson, Robert S. French, Christopher S. Wilson, Saman P. Amarasinghe, Jennifer M. Anderson, Steve W. K. Tjiang, Shih-Wei Liao, Chau-Wen Tseng, Mary W. Hall, Monica S. Lam, and John L. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. In *Proc. Fourth ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP)*, pages 31–37. ACM Press, 1994. http://suif.stanford.edu/suif/.

[Win01] Noel Winstanley. *Staged Methodologies for Parallel Programming.* PhD thesis, Department of Computing Science, University of Glasgow, April 2001.

[Wol87] Michel Wolfe. Iteration space tiling for memory hierarchies. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 357–361. SIAM, 1987.