Comparative Parallel Programming with PolyAPM using Abstract Parallel Machines

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

Abstract

A parallelising compilation consists of many translation and optimisation stages. The programmer may steer the compiler through these stages by supplying directives with the source code or setting compiler switches. However, for an evaluation of the effects of individual stages, their selection and their best order, this approach is not optimal.

We propose the following method for this special purpose. The compilation is cast as a sequence of program transformations. Each intermediate program runs on an Abstract Parallel Machine*, while the program generated by the final transformation runs on the target architecture. Our intermediate programs are all in the same language, Haskell. Thus, each program is executable and still abstract enough to be legible, which enables the evaluation of the transformation that generated it.*

PolyAPM provides a tree of APMs whose traversal specifies different combinations and orders of transformations. From one source program, several target programs can be constructed. Their run time characteristics can be evaluated and compared.

The goal of PolyAPM is not to support the construction of parallel application programs but rather to support a comparative evaluation of parallelisation techniques.

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 become 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. We will show examples of program properties, introduced by compiler transformations, that can be observed already in an intermediate APM program. Future work will enable a more detailed analysis of APM programs and relate these results to the run time behaviour of target programs running on parallel machines.

This paper is organised as follows: Section 2 gives a brief description of our PolyAPM framework and the APMs that we have designed. Section 3 presents an example derivation of a simple LU decomposition and compares two different branches after two alternatives of a design decision. Section 5 describes the potentials we envision for the PolyAPM approach and Section 6 concludes the paper.

2 Program development with Abstract Machines

The idea of stepwise refinement of specifications has long been prevalent in computer science. Trees are used to combine alternative sequences of design decisions. 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 eventually 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 intermediate program, we can have it implemented in software and even run our programs on it.

One specific kind of abstract machine has been described by O'Donnell and Rünger [12] 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 APMs and will base our work only loosely on the original APM.

2.1 PolyAPM

Rather than using the same APM for all transformations, like O'Donnell/Rünger, we choose 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 machine properties imposed on the program by successive transformations. The resulting sequence of intermediate programs is associated with a sequence of APMs. In general, there are fewer APMs than programs, since not every transformation introduces new machine requirements.

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. Actually, there are two graphs: one for the APMs themselves and one for the APM programs. 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 [10] within the LooPo project [9], 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 a parallelisation

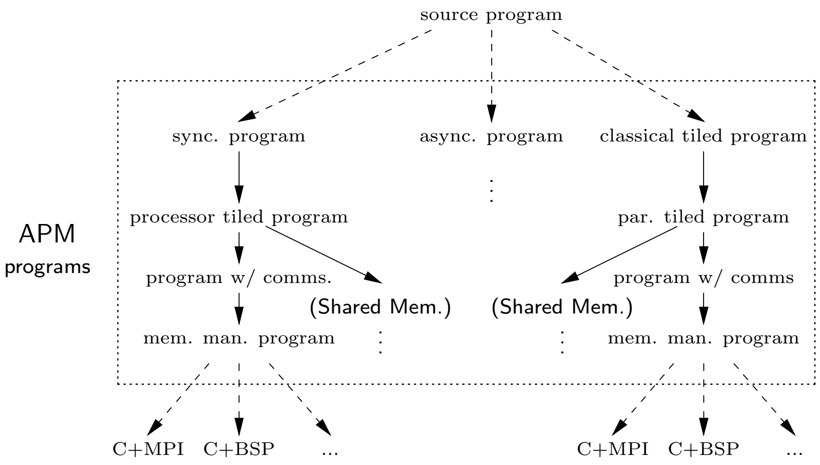

Figure 1. PolyAPM **Decision Graph (PDG), here only a sub-tree**

(foremost numerical computations), the Haskell implementation represents a natural "rephrasing" of the problem in just a slightly different language. This will be exemplified in Section 3. Second, as we use Haskell to implement the APMs, the APM programs' core is also a Haskell function that is being called by the APM interpreter. It should be avoided to have 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 a requirement.

2. Initial parallelisation of the sequential program. This requires the analysis of the problem to identify independent computations that can be computed in parallel – a process which might be done manually, or with the help of a parallelisation tool (as is the case within our example in Section 3). We have used LooPo [9] 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 a repetitive construct, e.g., 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: one processor loop which is parallel, one time loop which is sequential, and the loop body which is our atomic computation. If the outer loop is in time, we call the program *synchronous*, if it is in space, *asynchronous*. This motivates the corresponding branches of the PDG in Figure 1. The right branch for classical tiling is a special case where that 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 the source program's core computation). The program is subject to several transformations to adapt it to other APMs. This is the central aspect of PolyAPM and is discussed in more detail below in Section 2.2.

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, so that no optimisation of any APM program transformation is lost. Suitable target languages, among others, are C+MPI and C+BSP.

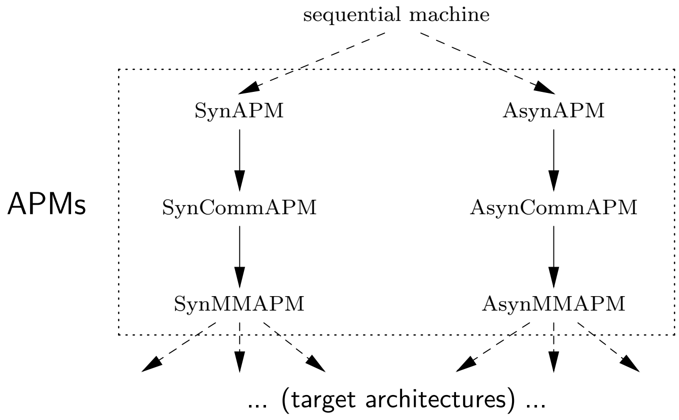

Figure 2. PolyAPM **Machine Tree**

2.2 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 which represent the parallel execution from the logical loops to be executed on the processors. Here, we deal only with a one-dimensional allocation. The model could be extended to incorporate multi-dimensional allocations.

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 processors' 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 divided into two transformations: First, while still maintaining a shared memory, we generate communication directives. As a second step, the memory is distributed, introducing the necessity of communication. 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 applied 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, while the effect missing communications would show 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 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 one for synchronous parallelism are possible.

```
lu_decomp:: Array (Int,Int) Float -> (Array (Int,Int) Float, Array (Int,Int) Float)
lu_decomp a = (l,u)
  where
   l = array ((1,1), (n,n))
     [ ((i,j), a!(i,j) - sum [ l!(i,k)*u!(k,j) | k <- [1..(j-1)]])
      | i <- [1..n], j <- [1..n] , j<=i ]

   u = array ((1,2), (n,n))
     [ ((i,j), (a!(i,j) - sum [ l!(i,k)*u!(k,j) | k <- [1..(i-1)]])/ l!(i,i))
      | j <- [2..n], i <- [1..n], i<j ]

   (_ , (n, _)) = bounds a
```

Figure 3. Haskell code for LU decomposition

In addition to the corresponding one for asynchronous parallelism, Figure 1 depicts also a typical sequence as employed by the tiling community [17].

3 Case Study

Our example algorithm is the LU decomposition of a non-singular square matrix $A = (a_{ij})\ ,\ (i, j = 1, \ldots, n)$.

The result consists of one lower triangular matrix $L = (l_{ij})$, with unit diagonal, and one upper triangular matrix $U = (u_{ij})$, such that $A = LU$.

L and U are defined recursively as follows [6]:

$$l_{ij} = a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj}, \quad j \leq i, \quad i = 1, 2, \ldots, n \tag{1}$$

$$u_{ij} = \frac{a_{ij} - \sum_{k=1}^{i-1} l_{ik} u_{kj}}{l_{ii}}, \quad j > i, \quad j = 2, \ldots, n \tag{2}$$

The Haskell implementation used for this example is shown in Figure 3.

Note the close relationship between the problem specification and the code. In particular, as the computations of L and U are mutually recursive, in most (i.e., strict) programming languages the programmer has to think about the data dependencies between L and U in order to find a sequential schedule. This additional "serialisation" would have to be undone in a subsequent parallelisation. However, this is not necessary with the given Haskell code. The lazy semantics of Haskell ensure a flow of computation as the data dependencies require, thus relieving the programmer of the burden to think about it.

In the following, we will present a sample derivation of the LU-program for a sequential loop nest for use with an SPMD message passing interface. We treat the above program as two statements within a two-dimensional index space spanned by i and j. The scalar product with index k is viewed as an atomic part of each computation.

3.1 Parallelisation

The parallelisation is done by first determining the data dependencies in the above program and then feeding these to the space-time mapping tools (i.e., the scheduler and allocator) of LooPo [9]. Details can be found elsewhere [5]. We also get bounds for the resulting loop nest. As we generate a synchronous APM program, the result is an imperative, strict loop program with array computations. A strict schedule for computing all array elements is given by the parallel schedule. The result is a two-dimensional loop nest where the computations of L and U have been inserted as two statements within the loop body.

The resulting APM language is embedded in Haskell, which enables us to use the two computation statements (after being subjected to the space-time mapping) directly. The resulting, synchronous APM program is depicted in Figure 4.

```
loop_syn = LP [(Seq, \([],[n])->0, \([],[n])->2*n ,2),
               (Par, \([t],[n])->t‘div‘2+1, \([t],[n])->n+1, 1)]
              (BD body_syn)

body_syn:: (LUmem, [Idx]) -> [Idx] -> (LUmem, [Idx])
body_syn  ((a,l,u),[n]) [t,p] = ((a, l_new, u_new), [n])
    where l_new = stmnt1 a l u (t,p,n)
          u_new = if (t+2)‘div‘2 == p
                  then u
                  else stmnt2 a l_new u (t+1,p,n)
          stmnt1 a l u (t,p,n) =
              l //  [((p,t‘div‘2+1) , a!(p,t‘div‘2+1)
                     - sum ([ l!(p,k)*u!(k,t‘div‘2+1)
                              | k <- [1..t‘div‘2]]))]
          stmnt2 a l u (t,p, n) =
              u // [(((t+1)‘div‘2, p), (a!((t+1)‘div‘2, p)
                                         - sum ([ l!((t+1)‘div‘2,k)*u!(k,p)
                                                  | k <- [1..(t-1)‘div‘2]]))
                    / l!((t+1)‘div‘2, (t+1)‘div‘2))]
```

Figure 4. Synchronous APM code of LU decomposition

The loop nest is two-dimensional, comprising an outer sequential loop ($0 \leq t < 2 * n$, stride 2) and an inner parallel loop ($\lfloor \frac{t}{2} \rfloor + 1 \leq p < n+1$, stride 1). The body function of SynAPM takes some *state*, consisting of memory and structure parameters, and a list of current values of all surrounding loops, to return an updated state. Here, the memory is defined as `LUmem`, a triple of the three arrays A, L and U. In general, an APM program must contain a loop structure, which includes a body function. Any type definitions are up to the convenience of the programmer. It may contain up to three additional functions, depending on the APM they are meant for. The function `synchronizeMem` is called after every time step and may be used perform memory reorganisation after all computations of a logical time step have been completed. In most cases, this function will just be the identity. The functions `generateMsg` and `updateMem` are used from SynCommAPM downwards. The first is called after every body execution and enables the creation of messages that send the just computed value(s) in a one-sided call (i.e., remote memory put), while the second is called by the communication system on the receiving side just to make the received values persistent in local memory. These functions have to be written by the programmer as they all require special knowledge about the APM program: `synchronizeMem` and `updateMem` manipulate the memory, which can be freely defined by the programmer, and `generateMsg` is obviously part of the problem specification anyway. This all together forms an APM program, written as a Haskell module, that can be executed with the corresponding APM interpreter (there is one for each APM machine), which is itself a Haskell module.

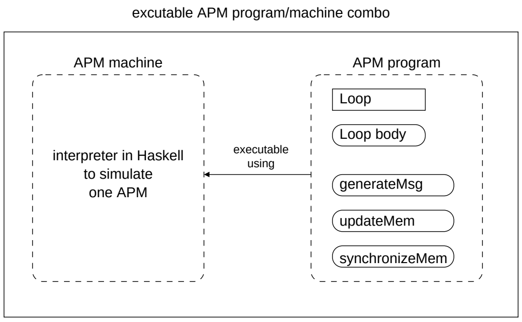

Figure 5. APM program structure

3.2 First Transformation: Processor Tiling

The first transformation within the APM framework according to the PDG (see Figure 1) is the *processor tiling* to reduce the number of parallel processors. Adding more detail to the depicted PDG, we observe that there are several tiling choices. We will explore two of them in this case study, namely block tiling and cyclic tiling. The two differ from the synchronous code in a change of the parallel loop's bounds and an additional inner sequential loop. Since these changes did not require new machine characteristics, the resulting, tiled programs also run on SynAPM.

Block tiled loop bounds with tile size defined as $\lfloor \frac{n-1}{\mathrm{maxp}} \rfloor + 1$:

```
loop_s_tl = LP [ (Seq, \([],[n])-> 0, \([],[n])-> 2*n, 2),
                  (Par, \([t],[n])-> 0, \([t],[n]) -> maxp, 1),
                   (Seq, \([t,rp],[n])->max (rp*(tilesize_p n)+1) (t‘div‘2+1),
                         \([t,rp],[n])->min ((rp+1)*(tilesize_p n)+1) (n+1), 1) ]
```

Cyclic tiled loop bounds:

```
loop_s_ctl = LP [ (Seq, \([],[n])-> 0, \([],[n])-> 2*n, 2),
                   (Par, \([t],[n])-> 0, \([t],[n]) -> maxp, 1),
                    (Seq, \([t,rp],[n])-> (t‘div‘2+1)+rp, \([t,rp],[n])-> (n+1), maxp) ]
```

These different processor tilings result in different parallel efficiencies of the program. Using profiling output of the APM interpreter, one can see that the original efficiency of 50% of the space-time mapped program is roughly maintained by the block-tiled derivative, but boosted to more than 90% by the cyclic tiled program. A shared memory OpenMP program written in C could be derived directly from the tiled programs, and the above results clearly suggest to use cyclic tiling.

3.3 Second Transformation: Generation of Communications

The second transformation, aiming at a point-to-point message passing library, is the generation of communications, while still keeping a shared memory. The changes of the previous tiling transformation only affected the loops. With the generation of the messages, we need to: change the memory data type into a state data type that combines memory and message queue and change the body function accordingly, add the implementation of `generateMsg` and `updateMem` and change the APM to SynCommAPM. Again, the changes of this transformation are clearly defined and quite localised, but due to space restrictions we will not display them here.

The network is assumed to be buffering and able to perform one-sided put communications. In the case of LU, we communicate along the data dependences which also had to be transformed by the space-time mapping. Originally, the dependences related L and U array cells with one another, which were later identified with virtual processors. After tiling the target virtual processors had to be identified with the real processors. This process reduced the number of message destinations and thus the number of messages. However, at this stage, message generation takes place after each body computation in order to send the newly computed value(s). Therefore, we have a communication structure in which more than one message may be sent from one real to another real processor. To reduce the number of messages on the network one should either have the sending routine aim at a message passing library that combines those messages automatically into a single one (like BSP), or have the message sending function do this explicitly.

3.4 Analysis

Now we are able to make statements about how the two tiled programs relate on the network. Profiling output of the SynCommAPM shows the following communication behaviour of cyclic versus block tiling with LU decomposition of an 8x8 matrix, as depicted in the Figure 6.

	block tiled	cyclic tiled
Computations	72	72
Communications on ..		
2 processors	16	64
4 processors	56	116
8 processors	140	140

Figure 6. Communication Behaviour

In short, one can say that the good processor utilisation of the cyclic tiled program backfires when it comes to communications. A lot of communications in the LU decomposition are along matrix rows and columns, so that with our allocation a lot of messages are to be sent to the next and all following virtual processors. In the block tiled program, the tiling was an order-preserving transformation on the order of virtual processors, so that in early stages of the computation quite some messages need not be sent to all busy processors. Furthermore, the number of busy processors decreases over time. Both observations contribute to a lower number of messages when compared to the cyclic tiling, whose the virtual processor order was not preserved by the tiling. Therefore, most messages have to be sent to all other physical processors, rendering them de facto broadcasts. In addition, the better processor utilisation over time means that until shortly before the end of the computation most processors participate and therefore communicate.

The above findings suggest that the block tiled program is more suitable for architectures using point-to-point communications with a high latency, which applies to most contemporary systems. In future work, we will refine the LU APM programs on our Linux SCI using MPI and BSP libraries. It is most likely that for a real speedup the ratio of tile computations to communications has to be improved, possibly by methods like *time tiling* [8].

In this simple example we have shown that some properties of the final parallel program can already be observed in the intermediate APM programs. This motivates the use of PolyAPM on a wider scale as outlined in Section 5.

4 Related Work

Systems to develop parallel programs fall into several categories: program transformation systems targeting abstract machines, compilation systems for general purpose languages with some support for parallel execution and dedicated automatic parallelisers.

4.1 Abstract Machine Models

John O'Donnell and Gudula Rünger have presented APMs [12], providing a starting point for others to work on parallel compilation using these.

Joy Goodman has extended the above work [7], included input and output via monads and investigated and formalised the decision making process.

Noel Winstanley also uses the APM methodology in his PEDL system [16]. He compiles array-based numerical programs to the parallel, imperative target language SAC. However, he uses a special restricted source language, tailored for his specific problem domain, and focuses on a high degree of optimisation and automation of the compilation.

4.2 Compilation Systems

Most compilers focus on the generation of efficient SPMD programs from source languages like Fortran. The source programs usually have to be augmented with parallelisation and data distribution annotations. However, some systems feature "automatic parallelisation" switches that enable either simple or semi-automatic parallelisation schemes. This group includes among others Adaptor [4], Polaris [3] and Parafrase [13]. A special role plays Bert77 [1], a Fortran77 source-to-source compiler that employs both static and dynamic parallelisation schemes, and focuses on performance prediction. A graphical user interface guides the semi-automatic process to improve the parallel performance based on a machine dependent cost model.

The SUIF [15] system serves as a compiler's workbench; the SUIF kernel defines an "intermediate representation" of a program between compiler phases and provides functions to access and manipulate it. SUIF is distributed with a set of example phases which includes a data dependency analysis and simple parallelisation techniques. However, parallelisation is not the foremost goal of the SUIF project.

In any case, each compiler has been designed with a rather fixed compilation process in mind. The compilation phases usually can be influenced by run time options, but more flexibility is rare. The most flexible parallelising compiler appears to be Bert77, which is claimed to be able to choose automatically between three different parallelisation schemes.

4.3 Parallelisation Systems

Automatic parallelisation systems still remain mostly a research topic. They are very specialised and work only on a restricted set of input programs. On this set, they usually provide an effective detection of parallelism. However, as they rely on a particular parallelisation method, their selection and ordering of transformations is mostly fixed. Examples are PAF [14], PIPS [2], OPERA [11] and LooPo [9].

5 Potential

We have presented an approach for the systematic development of parallel programs by applying a sequence of source-to-source transformations, which provides for a driven-by-need selection process of transformation techniques as well as means of evaluating and profiling the intermediate representation. Let us describe the potentials of this approach, motivated by the example in the previous section:

Evaluation of transformation effects: In contrast to classic compilation systems, on can easily observe the effects of a transformation in PolyAPM just by looking at the APM program code as well as retrieving simple profiling information off the interpreter and also by running it with different run time options an inputs using the corresponding APM interpreter. Especially with a long sequence of transformations, it may be very difficult to deduce from a final suboptimal compilation result which particular transformation has had the negative effect. With PolyAPM, one may identify and avoid this transformation and, in case of a manual compilation process, save oneself more unnecessary work by "cutting" the branch of the derivation tree and preempting the unsuitable sequence of transformations. This circumstance is closely related to the next potential.

Well-founded selection of transformations: Based on the fact that we can evaluate transformations, we can "deselect" bad ones and decide for a different transformation path. Furthermore, the modularity and extensibility of PolyAPM enables us to provide alternatives for transformations where traditional compilers just provide a fixed built-in transformation. A selection may be based on different criteria: the problem domain, the available parallel machine(s) (possibly of importance are, among others, number of processors, network topology, memory hierarchy), properties of a preferred message passing library, and so on. In addition, if the right choice is not evident, one has the opportunity to continue with a breadth-first search style of programming by selecting, transforming and evaluating a program, possibly followed by a backtracking step if the evaluation was not satisfactory.

A problem arises when two (or more) transformations at different levels of the PDG interact in such a way that the choice of the latter influences the validity of the evaluation of the former. In this case, a design decision cannot be based solely on the evaluation of the first transformation. Even if these interactions are not known beforehand, on has to be aware that they might exist.

Iterative automization towards a compiler: We have applied our PolyAPM transformations manually. But the system is designed to keep the changes to the program introduced by one transformation as small as possible. One reason for this is that they can then be easier performed automatically. This will lead to a potentially large number of automatic transformations. If all transformations along one path in the derivation tree are automatic, we have created a compiler. But, also a mix of manual and automatic transformations is possible, which allows for a stepwise development of a compiler. For special transformations an automatic solution might be to difficult and rarely needed, so that one is satisfied with the manual solution, rendering the entire system semi-automatic.

6 Conclusions

In summary, we view PolyAPM as a framework where code transformation techniques for a parallelising compiler can be implemented and evaluated. For a given problem, one can explore alternative transformations and determine which selection of transformations is best suited. It is even conceivable to make statements about which kind of machines one should use for a given class of problems.

7 Acknowledgements

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

References

[1] Bert77: Automatic and Efficient Parallelizer for FORTRAN, 2002. Paralogic Inc., Bethlehem, PA, USA, http://www.plogic.com.

[2] C. Ancourt, F. Coelho, B. Creusillet, F. Irigoin, P. Jouvelot, and R. Keryell. PIPS: A framework for building interprocedural compilers, parallelizers and optimizers. Technical Report A/289, Centre de Recherche en Informatique, Ecole des Mines de Paris, Apr. 1996. http://www.cri.ensmp.fr/~pips/.

[3] W. Blume, R. Eigenmann, et al. Polaris: Improving the effectiveness of parallelizing compilers. In K. Pingali et al., editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, 892, pages 141–154. Springer-Verlag, 1995.

[4] T. Brandes. *ADAPTOR Programmer's Guide, Version 6.0*, June 1998. Available via anonymous ftp from ftp.gmd.de as gmd/adaptor/docs/pguide.ps.

[5] N. Ellmenreich, M. Griebl, and C. Lengauer. Applicability of the polytope model to functional programs. In H. Kuchen, editor, *Proc. of 7th Int. Workshop on Functional and Logic Programming*, number 63 in Working Papers of the Institute of Business Informatics. Westf. Wilhelms-Universität Münster, 1998.

[6] C. F. Gerald. *Applied Numerical Analysis*. Addison-Wesley, 2nd edition, 1978.

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

[8] M. Griebl, P. Faber, and C. Lengauer. Space-time mapping and tiling – a helpful combination. *Concurrency, Practice and Experience*, 2003. To Appear. Preliminary version available as technical report MIP-0009 at Fakultät für Mathematik und Informatik, Universität Passau.

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

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

[11] V. Loechner and C. Mongenet. OPERA: A toolbox for loop parallelization. In I. Jelly, I. Gorton, and P. Croll, editors, *Proc. 1st Int. Workshop on Software Engineering for Parallel and Distributed Systems*, pages 134–145. Chapman & Hall, 1996. http://icps.u-strasbg.fr/opera/.

[12] J. O'Donnell and G. Rünger. A methodology for deriving abstract parallel programs with a family of parallel abstract machines. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *EuroPar'97: Parallel Processing*, LNCS 1300, pages 662–669. Springer-Verlag, 1997.

[13] C. Polychronopoulos, M. B. Girkar, M. R. Haghighat, C. L. Lee, B. P. Leung, and D. A. Schouten. The structure of Parafrase-2: An advanced parallelizing compiler for C and Fortran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'90)*, Research Monographs in Parallel and Distributed Computing, pages 423–453. Pitman, 1990.

[14] PRiSM SCPDP Team. Systematic construction of parallel and distributed programs. http://www.prism.uvsq.fr/english/parallel/paf/autom_us.html.

[15] R. P. Wilson et al. 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/.

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

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