Applicability of the Polytope Model to Functional Programs

Nils Ellmenreich, Martin Griebl and Christian Lengauer

May 29, 1998

Abstract

New areas of applications make the world of supercomputing more important than ever before. The programming of parallel machines remains difficult, even though new languages are being developed every year. They often follow the imperative paradigm, enriched with low-level parallel directives. To improve on this situation, we propose the use of functional languages, known for their ease of programming due to their high level of abstraction, and employ a specific parallelization method based on the polytope model, which has been successful for imperative scientific programs. We describe the prerequisites for adapting the polytope model to Haskell, a non-strict functional language. Different evaluation strategies are being considered and the question of how indexed data structures are to be dealt with in a functional language is examined. Finally, a parallelization of the LU decomposition algorithm exemplifies the method.

1 Introduction

In the past, many benefits of functional programming languages have been described and exploited, e.g., ability to reason, higher portability, higher expressiveness leading to fewer lines of code, and so on. One area which has long been proposed as a major argument in favor in functional languages, parallel programming, has up to now not achieved the role it was supposed to play. This is due to a problem which researchers in the early 80s did not anticipate: it is true that there is a lot of inherent parallelism in a functional program, but the problem is that there is too much of it and one has to contain and organize it. Dynamic control of parallelism, the dominant approach so far, seems not to provide all the instruments necessary.

In this paper, we take a new approach to parallel functional programming, driven by the demand of scientific computing. Our attempt is to gain as much static control over the parallel computation as possible. This area of application has been the driving force of high-performance computing in the past, both hardware and software. A typical application is an iterative algorithm using an indexed data structure, modeling matrices.

In our view, crucial for the acceptance is an implicit model of parallel computation, freeing the programmer from error-prone tasks of parallel programming and thereby smoothing the transition from sequential to parallel code, but imposing the need for automatic parallelizing compilers. In the imperative world, where most of the development of software for parallel processing takes place, implicit techniques for scientific programs can be formulated adequately within the polytope model [Len93].

1.1 Basics of the Polytope Model

The polytope model is being used so far for traditional imperative programs. It provides a framework for automatic parallelization of specific parts of a program. In particular, it deals with loop nests, usually for scientific programs.

The characteristics of a program fragment suitable for treatment with a parallelizer based on the polytope model are the following:

- Loop nests whose body consists of array assignments (we assume the single-assignment property for each array element).
- Loop bounds and array index expressions are typically affine and may contain other variables which are constant within the loop nest; no non-linear index expressions or indirect subscriptions are allowed.
- Other than assignments, only conditionals and procedure calls are analyzed in the loop body. The latter are treated as read and write accesses to all parameters, as far as the parallelization is concerned.

The task of determining the parallelism inherent in the loop nest and generating semantically equivalent, parallel code proceeds as follows:

- Data dependences have to be calculated. They associate accesses of different operations on the same array element. If such dependences exist, they impose a partial ordering on the operations. A sequential loop nest is in fact a specific traversal of the *index space*, a subset of $\mathbb{Z}^n$ with one point for each combination of loop variable values, which preserves this ordering.
- The task of "scheduling" is to find a function mapping all operations to points in time. This is also called the *time mapping*. The optimization aspect is to map every operation to the earliest possible time without violating the data dependences. If there is any parallelism in the program, there will be more than one operation scheduled at the same time.
- The dual mapping, the "allocation", assigns the operations to specific processing elements (PE). This is called the *space mapping*. One goal is to optimize data locality on the PEs, thus reducing communication. Ideally, one would like this to be adjusted to the schedule for optimal results, but in general this is not feasible.
- A code generation procedure performs a program transformation of the original source code into parallel target code by using the *space-time mapping*.

Intuitively speaking, the index space of the original loop nest is skewed by means of a coordinate transformation in such a way that there exist some dimensions to which no dependence arrows are parallel. These dimensions may be enumerated in parallel, the others are enumerated in sequence.

1.2 Prerequisites for Adaptation to FPLs

Functional languages differ fundamentally from imperative ones. Their semantics is based on the λ-calculus rather than the von Neumann model; this change of paradigm requires checking the work on the polytope model with respect to possible paradigm-based assumptions which might not apply anymore. Furthermore, new concepts such as the evaluation strategy have to be considered. It determines the run-time behaviour of a functional program and affects the expressiveness of the language, termination behaviour, degree of possible parallelism and more. Therefore, choosing the right evaluation strategy is crucial in a functional setting (see Section 2.2).

In order to adapt the polytope model to a functional language, we have to solve several problems:

- identify control structures in the functional language which can be dealt with in the model,
- determine the best possible evaluation strategy,
- examine efficient incremental array updates, both, sequential and parallel.

These tasks are discussed in the following sections.

2 Adapting the Model

This section elaborates on the prerequisites listed in Section 1.2.

2.1 Control Structures

Depending on the kind and structure of the imperative program, one has to choose the appropriate level of parallelism. The set of options include instruction-level, loop-level, intra-procedural and task parallelism. Scientific programs are typically loop nests with array assignment statements in the loop body. Other types of programs often cannot take advantage of specialized loop-based schemes and use combinations of instruction-level and one of the coarser-grain paradigms.

Now, the question arises of how this translates to the functional world. In this case, it is more difficult to categorize the abstraction level because all functional control structures can be used within the lowest-level expression up to the main function call. The selection of constructs which are potentially parallelizable include: subexpressions within an expression, independent function parameters, explicit recursion, predefined functions with recursive patterns, subexpressions in a data type constructor.

The best way of making a selection is to investigate the source of the problem: the set of imperative scientific programs containing loop nests often stems from mathematical recurrence relations on matrices. So the question is: if one implements a recurrence relation in a functional language, what are the constructs one should use? Recurrence relations define each element exactly once, so one would typically have an array constructor using a comprehension ranging over the array. In a lazy language, the actual evaluation order of the comprehension depends on the data dependences between array elements. For this interesting class of problems the task boils down to parallelizing the comprehension in order to construct the array. This approach is also being pursued by the NESL group [BCH+93].

For the mathematical treatment of array comprehensions in the polytope model we need to define the analogue of the imperative index space. We define the *index space* of an array definition as a subset of $\mathbb{Z}^n$ with one dimension for each comprehensions generator and the dimension's domain according to the range of the respective generator. The dimensionality of the array itself may be smaller than n due to the fact that the comprehension contains intermediate lists which are reduced in some way. Therefore, we define the *data space* as the domain of the defined array, thus being a projection of the index space. From now on, the term index space refers to the functional term if not stated otherwise.

As already mentioned, it is most interesting to view array comprehensions in the context of a lazy or, more generally, a non-strict semantics. See Section 2.2 for more details. In this case, even mutually recursive definitions are possible. See Section 3.1 for an example.

2.2 Evaluation strategy

The publications on functional programming do not agree on a common terminology. In order to compare evaluation strategies from different authors, we present our own set of definitions and point out where we differ from the respective authors' convention.

- *Strictness*: a function f with one parameter is called *strict* if $f(\perp) = \perp$. Functions with more than one parameter are strict if they are strict in every parameter.

- *Normal order* and *applicative order* evaluation are defined by the standard λ-calculus reductions strategies *leftmost-outermost* and *leftmost-innermost* [Rea89].
- *Lazy evaluation* is defined as the reduction strategy *call-by-need*, which in turn is normal order reduction with sharing of common subexpressions (based on graph reduction).
- *Eagerness* is defined as a property of many evaluation orders, such that all actual parameters will be evaluated before the function returns its result. In particular, unnecessary computations may be performed.
- *Fair* is a parallel evaluation strategy which has a fair scheduling policy and, thus, prevents starvation. The opposite is *unfair*.

<table>
<tr><th>Strictness semantics</th><th>Evaluation strategy</th><th>Par/Seq</th><th>Eager</th><th>Speculation</th><th>Data structures</th></tr>
<tr><td rowspan="2">strict</td><td>applicative order</td><td>Seq</td><td rowspan="2">Yes</td><td rowspan="2">Yes</td><td rowspan="2">finite</td></tr>
<tr><td>eager-parallel-unfair</td><td>Par</td></tr>
<tr><td rowspan="3">non-strict</td><td>eager-parallel-fair</td><td>Par</td><td>Yes</td><td>Yes</td><td>circular</td></tr>
<tr><td>lazy</td><td rowspan="2">Seq</td><td rowspan="2">No</td><td rowspan="2">No</td><td rowspan="2">infinite</td></tr>
<tr><td>normal order</td></tr>
</table>

Table 1: Functional evaluation strategies

An overview of the properties of common evaluation strategies is given in Table 1. In the literature, the terms used are not generally agreed upon. Let us discuss the table in more detail.

The two major categories which separate all strategies are strict and non-strict. *Strict* evaluation has a potential for parallelism mainly in the evaluation of function arguments. With strictness, mutually recursive and self-referencing data structures are not possible, because during the construction of a data item the evaluation of the self-reference would have to be delayed – otherwise there would be an infinite loop. With strictness, there is no such delay. Therefore, all substructures have to be independent and can be computed in parallel but, due to the lack of data dependences among them, each single item is often very simple, so that it might not pay to make everything parallel. The lack of mutual recursion makes it hard to implement recurrence relations.

The alternative is *non-strictness*, where functions may terminate even if the evaluation of some arguments do not terminate. The reason is that a strict function will have to evaluate all arguments before returning the result while a non-strict function does not have to. Non-strictness is a source of greater expressiveness, e.g., it affects directly the type of data structures that can be dealt with. Strict functions evaluate all parameters to normal form, which is the reason why they can only deal with finite data structures. Non-strict functions may not evaluate everything; thus, they are able to handle finite parts of possibly infinite data structures. This greater expressiveness is the driving force behind the design of non-strict languages.

First, let us have a look at the strict orders. *Applicative order*, also known as *call-by-value*, is a well defined sequential strategy. Some authors use *eager evaluation* as a synonym for it, but in our terminology it is just one of several eager orders. Due to eagerness there is a possibility of speculative computations.

One possible strict and parallel strategy is *eager-parallel-unfair*, also known as *parallel-innermost* [Rea89]. It could be described as evaluating all function parameters simultaneously in a strict manner without taking care of load balancing. This affects the termination behaviour and is in this respect not different from applicative order. The possible parallelism is restricted to those redexes which do not contain another inner redex. It is called *unfair*, since strictness forces the evaluation of unnecessary and possibly non-terminating computations.

An improvement with respect to parallelism has been made by combining non-strictness with eagerness to *eager-parallel-fair*, which is non-strict, but not necessarily lazy. This strategy is called *lenient* by Traub [Tra91] and has been used in the languages Id [Nik91] and pH [NAH+95]. The main goal is to keep the expressive power of non-strictness, i.e., more complicated data dependences, while improving the degree of parallelism with respect to lazy evaluation at the expense of the impossibility of infinite data structures. The reduction strategy is not defined in terms of the λ-calculus; instead, these languages are being given an operational semantics which resembles the data flow paradigm: expressions can be evaluated as soon as their inputs are available. This ensures non-strictness, but not laziness. In particular, function bodies can be evaluated while not all arguments are in normal form. This enables recursive self-references and consequently circular data structures. But eventually all arguments are being reduced, so that infinite data structures are not possible. An example of a circular data structure is a circular list, in Haskell notation `a = 1:2:a`. An example of an infinite data structure is the list of all integers, in Haskell `b = [1..]`.

Finally, *lazy* and *normal order* reduction are again defined in terms of the λ-calculus. They are sequential – not eager – and therefore no (possibly unnecessary) speculative computations take place. As a result, infinite data structures can be defined of which finite parts may be computed. The difference between both is lazy's added sharing of subexpressions for efficiency reasons.

Lazy evaluation is the standard for sequential non-strict languages like Haskell [PHe96]. Parallel versions use either a variation of laziness combined with explicit parallelism annotations (GPH [THLJ98] and Clean [BvEvL+87]) or other parallel, non-strict strategies like pH [NAH+95], which uses eager-parallel-fair or lenient evaluation. In scientific computing, the main data structure is the fixed-size matrix resp. array, either sparse or dense. Therefore, the drawback of excluding infinite data structures is no restriction, so that lenient evaluation would suffice for our purposes.

2.3 Indexed Data Structures

In Section 1, we have motivated the need for parallelization of scientific programs. Since scientific computing is based on mathematics, the programmer often has to find a representation of matrices in the programming language he uses. This requires n-dimensional integer indexing and often elementwise indexed access to the data. Two different access patterns of matrices can be observed: one constructs the entire matrix step by step without ever changing a once-written value, the other iterates over temporary values and overwrites values in the matrix. In contrast to the imperative paradigm, where an array element corresponds to nothing but a memory location, these two patterns must be handled very differently in functional programming. This is due to constraints imposed by the language semantics and is explained in more detail below.

One frequently stated advantage of functional languages is the comparative ease of proving program properties. One prerequisite for this is the *single-assignment* property, i.e., every *name* is associated with a *value* not more than once. In particular, this rule forbids *reassignment* (or better: rebinding) of values to names, as is common in an imperative program. The reason is that one often wants to have a *referentially transparent* language, in which both sides of an equation can be used interchangeably and are, in fact, interchanged in proofs. Formally, a language L is referentially transparent, if and only if: $E_1 = E_2 \Longrightarrow E = E[E_1 := E_2]$ for all expressions E, E_1 and E_2 in L. Therefore, the value of a name must remain constant within its scope. This requirement is not generally accepted; there are functional languages which, under certain circumstances, allow reassignment and pay for it with a less elegant semantics; ML [HMT88], LISP [MAE+64] and Id [Nik91] are examples.

Now, algorithms can be divided into three kinds with respect to their use of arrays.

1. All elements are constructed independently, so that the matrix can be written in one step and has the single-assignment property.

2. Each element is defined exactly once, but for their calculations the final values of other elements are needed. These elements have the single-assignment property, but the matrix cannot be written in one step. One would need deferred or lazy evaluation to construct it.

3. Array elements are repeatedly overwritten.

From the functional programming point of view, the first kind is the easiest: arrays are computed once and from then on treated as read-only objects. The second requires a non-strict (e.g., lazy) evaluation – otherwise the single-assignment property is lost. The third gives rise to major implementation problems: for semantic reasons, most functional languages disallow this. The usual way of avoiding this restriction is to require the programmer to define a new array for every update which differs from the previous one only in the updated element, which brings the algorithm back to one of the first two kinds. The implementation has two options — *incremental* updates are stored as differences to the original array, increasing the access time, whereas *monolithic* arrays require a complete copy of the source, resulting in increased memory consumption but $O(1)$ access time. Both updates create new arrays. Haskell uses monolithic arrays.

Several optimization techniques are being used to improve this situation. The question is whether the array's use is single- or multi-threaded. In *single-threaded* use, once an array has been updated, the old version is not being used anymore. This is also the case in imperative languages; in fact, here the old copy cannot be used anymore since the old value has been overwritten destructively. In functional languages like Haskell, the updated array requires a new name, so that the old copy is still around and can be used, e.g., for a different update. The latter way to use arrays is called *multi-threaded*. Now, if the compiler has a means of detecting single-threaded use – or, maybe, the language even enforces it – then the update can be done *in place*, without copying the array. Furthermore, even in Haskell, the programmer might enforce single-threaded use by encapsulating the array in a monad [Wad92] . An appropriate compiler could take advantage of this situation. On the other hand, traditional computer architectures are not capable of a true $O(1)$ update of a multi-threaded array [O'D93]. Other possibilities include sophisticated data structures with amortized update costs of close to $O(\log n)$ in the worst case [OB97].

Another kind of overhead of most arrays in functional languages is incurred by of boxed types. In contrast to C, where an integer array is represented by a contiguous amount of memory consisting of integer values, each data element in, e.g., Haskell is *boxed*, i.e., has one additional level of redirection to be able to store values like $\perp$. To make scientific computations which, more often than not, deal with integers and floats more efficient, the use of *unboxed* data types is necessary. Sometimes such data types are provided as compiler-specific extensions. A different approach is to separate structure and type information from the actual data leading to an efficient unboxed representation [JS98].

An alternative to using an array at all is the *indexed list* [EL97]. This less general approach than the array can be sufficient in lots of cases and has an efficient implementation. The basic idea is that of a list data type whose *cons* operation appends at the end and whose list elements can be retrieved by their respective index in $O(1)$.

This leaves us with the question of which possibility to choose. Standard mathematical algorithms do not require multi-threadedness and unboxed types are necessary for efficiency reasons. It seems that, for this specific purpose, a new data structure is needed. This could be the indexed *snoc* list [GG97], if applicable. In all other cases, we propose a new array-like data structure with only few, unboxed data types, whose single-threaded use should be enforced by the compiler. The implementation, however, should make in-place updates for efficient access.

```
lu_decomp:: Array (Int,Int) Float -> (Array (Int,Int) Float,
                                      Array (Int,Int) Float)
lu_decomp a = (l ,u)
  where
   l:: Array (Int,Int) Float
   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 (Int,Int) Float
   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 1: Haskell code for LU decomposition

3 The Polytope Model in Action

We present an example parallelization in order to give a feel for how the anticipated method might work. For some of the practical calculations, we used the loop parallelizer LooPo [GL96], a test suite for different parallelization strategies based on the polytope model. It can be used to parallelize loop nests written in Fortran or C. Some steps of the parallelization process are language-independent; the respective modules can be used for our purposes.

3.1 The Example: LU-Decomposition

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 $U = (u_{ij})$, whose product must be A.

L and U are defined constructively as follows [Ger78]:

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

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

The Haskell implementation used for this example is shown in Figure 1.

3.2 Dependence Analysis

Data dependences between computations impose a partial order which any evaluation order must follow. The partial order is used to identify independent computations which can be executed in parallel. The aim of dependence analysis is to obtain this partial order by analysis of the source code. In the following, we present a simple algorithm to determine the dependences between array elements in array computations of a Haskell program.

Array computations in Haskell are often specified with the `array` construct. The array elements are usually defined by an array comprehension. Due to the fact that data type definitions can be mutually recursive, a set of array definitions may depend on each other. We define two arrays a and b to be in relation R, i.e., aRb iff a value from b is being referenced in the definition of a. Then, the weakly connected components with respect to R are the sets of arrays, in which each array is either the source or the destination of a dependence. The computations of the arrays in a component are parallelized together.

The first task is to determine these array component sets, given a Haskell program. This is done by the pseudo-algorithm in Figure 2.

Input: – Haskell-program with array definitions
Output: – Array component sets S_i, $i \in \mathbb{N}$

1. Construct a list L of all arrays in the program.
2. `while` there exists an unmarked array A in L `do`
3. Construct the weakly connected component S_A of arrays which includes A.
4. Mark all arrays of set S_A in list L
5. `output` all S_i

Figure 2: Array component set determination

The second task is to determine all dependences within a set. They will be used later on for the parallelization of all computations within the set. The pseudo-algorithm for the dependence analysis is presented in Figure 3.

Input: – Haskell-program with array definitions
– Component set S
Output: – Set of all dependences with index spaces of set S

1. `foreach` array A in S
2. `foreach` array A' of S used in the definition of A
3. construct a dependence A' $\rightarrow$ A; add it to set D, together with appropriate index space.
4. `output` D

Figure 3: Dependence Analysis Algorithm

The determination of the index space in step 3 has to consider possible boolean restrictions of generators in Haskell's list comprehension. The original polytope model [Len93], is restricted to index spaces with affine bounds. For simplicity, we use this model in the context of this paper, i.e., the boolean predicates may only be affine relations of index variables. Extensions of the polytope model can deal with general `IF` statements [Col95], which correspond to arbitrary boolean predicates in our context.

Applying the dependence algorithm to the LU example, we end up with one component set S comprising A and the two mutually recursive arrays L and U, both reading A. The set of dependences is presented in Figure 4.

This set of dependences describes a partial order on the index space of L and U. The figure contains, for each array, the referenced array elements (of itself and others), the restriction imposed on the original array's domain definition (an affine relation) and the restricted index space of the

Arr	No	Source	Dest.	Restr.	Restricted Index Space
L	1	$l(i,k)$	$l(i,j)$	$j \leq i$	$(i,j,k) \in \{1,\ldots,n\} \times \{1,\ldots,i\} \times \{1,\ldots,(j-1)\}$
	2	$u(k,j)$	$l(i,j)$		
U	3	$l(i,k)$	$u(i,j)$	$i < j$	$(i,j,k) \in \{1,\ldots,(j-1)\} \times \{2,\ldots,n\} \times \{1,\ldots,(i-1)\}$
	4	$u(k,j)$	$u(i,j)$		
	5	$l(i,i)$	$u(i,j)$		

Figure 4: Dependences in the LU example

dependence. To get an idea of the structure of the dependences, Figure 5 contains a graphical presentation of the data space with L's dependences on the left and U's dependences on the right. The numbers next to the different types of lines in the legend correspond to the dependence numbers in Figure 4. The reason for the different dimensionality of data and index space is in our case the intermediate list generated by k, which is summed up. Dependences and index spaces are input for a scheduling algorithm which is described in the next section.

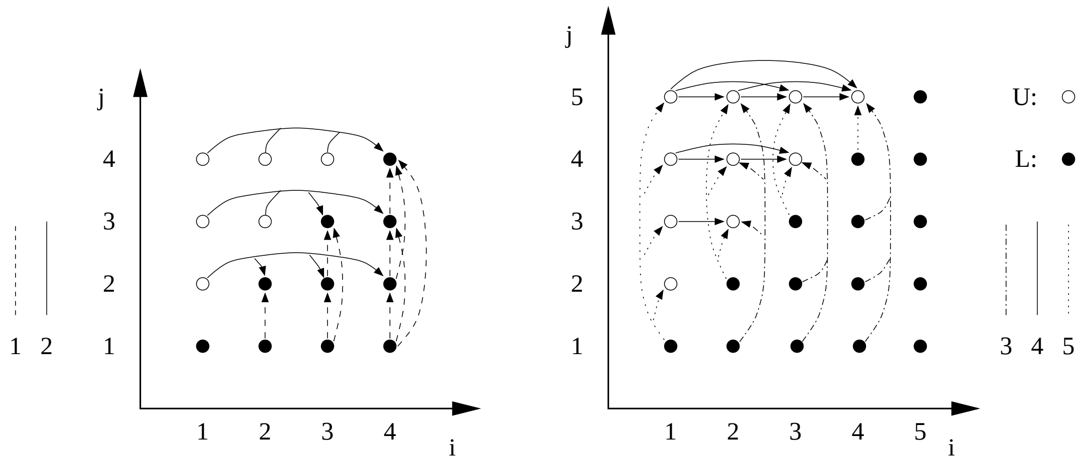


Figure 5: Dependences between array elements

3.3 Schedule and Allocation

Using the Feautrier scheduler [Fea92] of LooPo [Wie95] to get a schedule for the dependences mentioned above, we obtain:

$$\theta_l(i,j) = 2*(j-1) \quad (3)$$

$$\theta_u(i,j) = 2*(i-1)+1 \quad (4)$$

This schedule honors the fact that the definitions of L and U are mutually recursive, so that their overall computation is interleaved. Note that, at each point in time, several computations can be performed, e.g., at logical time $2*(j-1)$ we can compute $l(i,j)$ for all i.

The mapping of computations, which are performed at the same time, to virtual processors is defined by the allocation function. Finding a valid function is not difficult, since every mapping from the set of parallel computations to the natural numbers will do. The difficulty is finding a sensible function which minimizes the number of communications. This is done by placing dependences on single processors, i.e., allocating a computation on the same processor as a previous computation it depends on. Then, the data item can simply stay in the local memory of the processor. Up to now, no provably optimal algorithm for generating an allocation has been found, but some heuristic algorithms have been proposed [Fea94, DR95].

We use LooPo's Feautrier allocator [Wie95] to compute suitable allocation functions for the LU example:

$$\sigma_l(i,j) = i \tag{5}$$

$$\sigma_u(i,j) = j \tag{6}$$

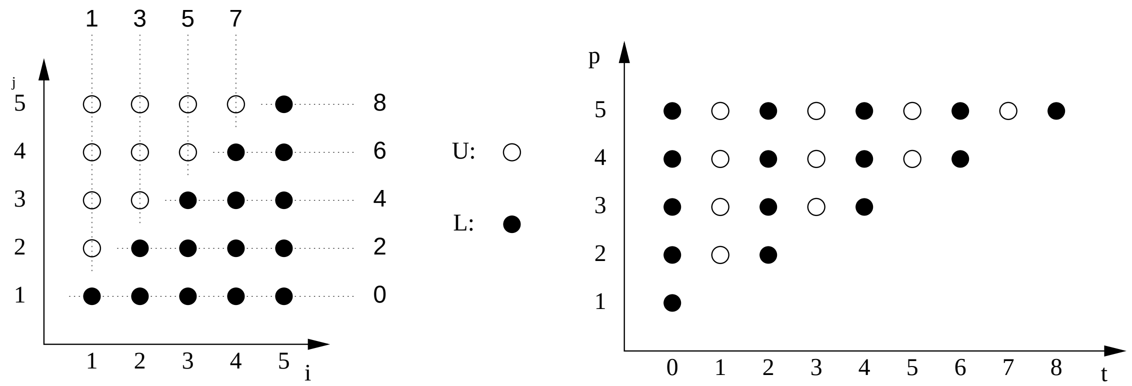
Now we combine schedule and allocation to a single transformation matrix, which is used to perform a coordinate transformation of the index pairs (i,j) into (t,p), which corresponds to the enumeration of time and processors. The scheduler generates a schedule for each computation so that we have two matrices, one for L and one for U. Generally, the relation $T \cdot \binom{i}{j} + \vec{d} = \binom{t}{p}$ holds, in this case, denoting a mapping from the index space $\{(i,j) \mid (i,j) \in \mathbb{N}_+^2\}$ to the target space $\{(t,p) \mid (t,p) \in \mathbb{N}^2\}$. We obtain:

$$T_L = \begin{pmatrix} 0 & 2 \\ 1 & 0 \end{pmatrix} \; ; \; \vec{d}_L = \begin{pmatrix} -2 \\ 0 \end{pmatrix} \; ; \; T_U = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \; ; \; \vec{d}_U = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

This matrix presentation combines the data necessary for the coordinate transformation.

To see the effect on the index space, Figure 6(a) presents the computations of L and U in a single index space (their domains do not intersect, so that they could even be stored in a single matrix). In this example, we choose a value of 5 for n. The data points which are independent of each other lie on the same dotted schedule line. The target space in Figure 6(b) depicts the points after the coordinate transformations. Data items with the same schedule now have the same value t, meaning that they are computed at the same time.

Thus, as a result of the parallelization, the computation of the LU decomposition has been accelerated from 25 to 9 virtual time units (n^2 compared to $2n-1$), where one unit is the time to compute a single data item. This time unit depends on the level of abstraction we have chosen here. In the Haskell program in Figure 1, each computation of L and U contains the summation of a list whose length is $O(n)$. Taking this into account, a refined parallel solution requires about $3n$ time units, which corresponds to other results [Che86].



(a) Index space of L and U (b) Target space of L and U

Figure 6: Before and after the space-time mapping

3.4 Code Generation

So far, in the process of parallelizing a program, we have analyzed the source program to get a set of parallelizable array definitions, we have determined the data dependences between array elements and we have found schedule and allocation functions to specify when and where computations should be performed. The final task is code generation, i.e., generating a semantically equivalent parallel target program by using schedule and allocation to specify the parallel behaviour explicitly.

Our work on code generation has only just begun. For the target language, we had several options:

- explicitly annotated functional languages based on parallel graph reduction,
- imperative languages with parallelism annotations like Fortran90,
- imperative languages with low-level libraries like C plus MPI,
- other high-level program transformation techniques, based on, e.g., abstract machines.

Our aims were to choose a framework which is flexible enough to be adapted to the specific needs of our scheme and allows for sufficient possibilities to tune the result. We believe that a suitable collection of abstract parallel machines (APM) [OR97] fits these needs best. The idea is to perform a sequence of program transformations, each being a bit more specific, to convert an abstract presentation of an algorithm into low-level executable code for a parallel machine.

We plan to design APMs for subsets of Haskell enriched with low-level directives for parallelism. The process of transformation should eventually be mostly automatic, but the user should always have the option of superceding the automatic process. At the end of the transformations, we aim at a C plus MPI interface.

4 Related Work

There have been a lot of other efforts to combine functional and parallel programming for scientific computing. Probably most widely known is SISAL [BOCF92], a strict and lately even higher-order language developed at Lawrence Livermore National Labs, which was designed to motivate former FORTRAN programmers to use a functional language and thereby gain expressive power without losing too much efficiency. However, SISAL lacks a strict semantics and several features of a modern functional language like an advanced type system with polymorphism.

NESL [BCH^{+}93] is a project at CMU to devise a functional language which exploits implicit parallelism contained in nested list comprehensions and some built-in functions. Our approach is more general in that it can parallelize different expressions together and is not restricted to lists.

Alpha [Wil94] is a restricted functional language aimed at the synthesis of regular architectures. The input program, based mainly on recurrence equations, is refined in transformational steps using, among others, space-time mappings in the fashion of the polytope model. Our approach aims at scientific programs rather than VLSI.

Glasgow Parallel Haskell [THLJ98] exploits explicit parallelism of annotated Haskell programs using *strategies*, which describe the possibly parallel evaluation behavior of the annotated expression. This is explicit task parallelism rather than our implicit and, possibly, finer grained parallelism.

The MIT group of Arvind and Nikhil first designed Id [Nik91], a data flow language whose main contributions were the introduction of I- and M-structures, which are used to provide efficient, indexed data structures for a parallel environment at the cost of destroying some semantic properties of the language. More recent work adapts the ideas of Id to a Haskell dialect, pH [NAH+95] (parallel Haskell). The language differs from Haskell in replacing lazy with lenient evaluation. The main difference to our work is that they add impure data structures for parallelism, whereas we keep the purely-functional semantics of Haskell.

5 Conclusion

For the restricted case of scientific array computations, the automatic parallelization process using the polytope model can be adapted to the functional paradigm. It remains to be combined with other techniques for general-purpose use – this can be done in the APM framework.

6 Acknowledgments

The authors would like to thank John O'Donnell for reading the draft and lots of helpful comments, especially on Section 2.2, Björn Lisper for fruitful discussions and the DAAD for travel funding to both partners.

References

[BCH+93] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *Principles and Practices of Parallel Programming*, pages 102–111, 1993.

[BOCF92] A. P. Wim Böhm, Rodney R. Oldehoeft, David C. Cann, and John T. Feo. *SISAL Reference Manual, Language Version 2.0.* Colorado State University – Lawrence Livermore National Laboratory, 1992.

[BvEvL+87] T. Brus, M. van Eekelen, M. van Leer, M.J. Plasmeijer, and H.P. Barendregt. Clean – a language for functional graph rewriting. In *Proc. Third International Conf. on Functional Prog. Languages and Comp. Architectures (FPCA'87)*, Lecture Notes in Computer Science 274, pages 364–384. Springer-Verlag, 1987.

[Che86] Marina C. Chen. Placement and interconnection of systolic processing elements: A new LU-decomposition algorithm. Technical Report YALEU/DCS/RR-498, Dept. of Computer Science, Yale University, October 1986.

[Col95] Jean-François Collard. *Parallélisation automatique des programmes à contrôle dynamique.* PhD thesis, Université Paris VI, 1995.

[DR95] Michéle Dion and Yves Robert. Mapping affine loop nests: New results. In *Lecture Notes in Computer Science 919*, pages 184–189. Springer-Verlag, 1995.

[EL97] Nils Ellmenreich and Christian Lengauer. On functional scientific computing. In John T. O'Donnell, editor, *Proc. Glasgow Functional Programming Workshop.* Department of Computer Science, University of Glasgow, 1997. To appear in electronic form.

[Fea92] Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, October 1992.

[Fea94] Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.

[Ger78] Curtis F. Gerald. *Applied Numerical Analysis*. Addison-Wesley, 2nd edition, 1978.

[GG97] Alfons Geser and Sergei Gorlatch. Parallelizing functional programs by generalization. In M. Hanus, J. Heering, and K. Meinke, editors, *Algebraic and Logic Programming (ALP'97)*, Lecture Notes in Computer Science 1298, pages 46–60. Springer-Verlag, 1997.

[GL96] Martin Griebl and Christian Lengauer. The Loop Parallelizer Loopo. In Michael Gerndt, editor, *Proc. 6th Workshop on Compilers for Parallel Computers (CPC'96)*, volume 21/1996 of *Konferenzen des Forschungszentrums Jülich*. Forschungszentrum Jülich GmbH, 1996.

[HMT88] Robert Harper, Robin Milner, and Mads Tofte. The Definition of Standard ML, Version 2. Technical Report ECS-LFCS-88-62, Laboratory for Foundation of Computer Science, University of Edinburgh, 1988.

[JS98] C. Barry Jay and Paul A. Steckler. The functional imperative: shape! In Chris Hankin, editor, *Programming Languages and Systems: 7th European Symp. on Programming (ESOP'98)*, Lecture Notes in Computer Science 1381, pages 139–53. Springer-Verlag, 1998.

[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.

[MAE+64] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *Lisp 1.5 Programmer's Manual*. MIT Press, 1964.

[NAH+95] Rishiyur S. Nikhil, Arvind, James Hicks, Shail Aditya, Lennart Augustsson, Jan-Willem Maessen, and Yuli Zhou. pH Language Reference Manual, Version 1.0. CSG-Memo-369, MIT, Computation Structures Group, 1995.

[Nik91] Rishiyur S. Nikhil. Id (Version 90.1) Reference Manual. CSG-Memo-284-2, MIT, Computation Structures Group, 1991.

[OB97] Melissa E. O'Neill and F. Warren Burton. A new method for functional arrays. *Journal of Functional Programming*, 7(5):487–513, September 1997.

[O'D93] John O'Donnell. Data parallel implementation of extensible sparse functional arrays. In *Parallel Architectures and Languages Europe (PARLE'93)*, Lecture Notes in Computer Science 694. 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*, Lecture Notes in Computer Science 1300, pages 662–669. Springer-Verlag, 1997.

[PHe96] John Peterson and Kevin Hammond (editors). Report on the Programming Language Haskell, A Non-Strict Purely Functional Language (Version 1.3). Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, May 1996.

[Rea89] Chris Reade. *Elements of Functional Programming*. International Computer Science Series. Addison-Wesley, 1989.

[THLJ98] Phil Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon Peyton Jones. Algorithm + Strategy = Parallelism. *J. of Functional Programming*, 8(1), January 1998.

[Tra91] Kenneth R. Traub. *Implementation of Non-Strict Functional Programming Languages.* Research Monographs in Parallel and Distributed Computing. Pitman Publishing, 1991.

[Wad92] Philip Wadler. Monads for functional programming. In *Lecture Notes for Marktoberdorf Summer School on Program Design Calculi*, pages 233–264. Springer-Verlag, August 1992.

[Wie95] Christian Wieninger. Automatische Methoden zur Parallelisierung im Polyedermodell. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[Wil94] Dennis K. Wilde. The Alpha Language. Technical Report 999, IRISA, January 1994.