JFP **15** (3): 353–401, 2005. © 2005 Cambridge University Press
doi:10.1017/S0956796805005538 Printed in the United Kingdom

Shared memory multiprocessor support for functional array processing in SAC

CLEMENS GRELCK

Institut für Softwaretechnik und Programmiersprachen, Universität zu Lübeck, Ratzeburger Allee 160, 23538 Lübeck, Germany
(*e-mail:* grelck@isp.uni-luebeck.de)

Abstract

Classical application domains of parallel computing are dominated by processing large arrays of numerical data. Whereas most functional languages focus on lists and trees rather than on arrays, SaC is tailor-made in design and in implementation for efficient high-level array processing. Advanced compiler optimizations yield performance levels that are often competitive with low-level imperative implementations. Based on SaC, we develop compilation techniques and runtime system support for the compiler-directed parallel execution of high-level functional array processing code on shared memory architectures. Competitive sequential performance gives us the opportunity to exploit the conceptual advantages of the functional paradigm for achieving real performance gains with respect to existing imperative implementations, not only in comparison with uniprocessor runtimes. While the design of SaC facilitates parallelization, the particular challenge of high sequential performance is that realization of satisfying speedups through parallelization becomes substantially more difficult. We present an initial compilation scheme and multi-threaded execution model, which we stepwise refine to reduce organizational overhead and to improve parallel performance. We close with a detailed analysis of the impact of certain design decisions on runtime performance, based on a series of experiments.

1 Introduction

Functional programming languages are generally considered well-suited for parallelization. Program execution is based on the principle of context-free substitution of expressions. Programs are free of side-effects and adhere to the Church-Rosser property. Any two subexpressions without data dependencies can be executed in parallel without any further analysis. Unfortunately, parallel execution inevitably incurs overhead for synchronization and communication between cooperating threads or processes. In practice, it turns out that identification of concurrent subexpressions whose parallel execution actually pays off can be almost as difficult as parallelization of imperative code (Hammond, 1994).

Numerous approaches have been developed to address this problem. In one way or another, they provide the execution machinery with hints of where and how to exploit concurrency (Cole, 1989; Nöcker *et al.*, 1991; Darlington *et al.*, 1993; Bratvold, 1993; Bülck *et al.*, 1994; Trinder *et al.*, 1998; Trinder *et al.*, 1999; Hammond & Portillo, 2000). Unlike annotations in imperative settings, e.g. OpenMP (Dagum & Menon,

1998), they solely affect runtime performance, but not correctness of results. Still, the goal to obtain parallel code from sequential specifications by simple recompilation is not achieved.

Other approaches provide programmers with more or less explicit means for process or thread management synchronization, and communication, (Cooper & Morrisett, 1990; Reppy, 1991; Bailey & Newey, 1993; Jones *et al*., 1996; Breitinger *et al*., 1997; Kelly & Taylor, 1999; Serrarens, 1999). However, to the same extent as they provide control over parallel execution, these approaches also introduce the pitfalls of traditional parallel programming, e.g. deadlocks or race conditions.

Following their sequential host languages, all the approaches mentioned so far focus on lists or on algebraic data types in general. They are suitable tools for the parallelization of symbolic computations (Trinder *et al*., 1996; Loidl *et al*., 1999), where they benefit from the expressive power of their host languages. However, classical domains of parallel computing like image processing or computational sciences are characterized by large arrays of numerical data, not lists (Bailey *et al*., 1991).

Unfortunately, most functional languages are not well-suited for high-performance array processing. Notational support for multi-dimensional arrays is often rudimentary. Even worse, sequential runtime performance in terms of memory consumption and execution times fails to meet the requirements of numerical applications (Hartel & Langendoen, 1993; Hartel *et al*., 1996; Hammes *et al*., 1997). Their standards are set by existing implementations written and hand-optimized in low-level imperative languages like C or FORTRAN.

Parallelization is no remedy to poor sequential performance. This would require a completely implicit approach similar to a compiler optimization in addition to free availability of multiprocessor resources. In practice, however, parallelization does require additional coding, and people are usually unwilling to pay for a supercomputer only to obtain a similar runtime performance as with an imperative implementation on a uniprocessor desktop machine. At least when addressing classical domains of parallel computing, parallelization is hardly worth the effort as long as the performance of established solutions is out of reach in sequential execution.

Low sequential performance also confronts research on parallelization techniques with a specific pitfall. The ratio between time spent in synchronization/communication operations and time spent in productive code is shifted in favour of productive code. This effect makes parallelization techniques look more efficient than they actually are. The true impact of design decisions on performance is disguised, leading to false assessments.

Reasons for the poor runtime performance of general-purpose functional languages in processing multi-dimensional arrays are manifold. Certainly, all high-level language features like polymorphism, higher-order functions, partial applications, or lazy evaluation contribute some share of the overhead, but array-specific pitfalls exist as well. Conceptually, functions consume argument values and create result values from scratch. For small data items, such as list cells, this can be implemented fairly efficiently. However, operations which "change" just a few elements of a large monolithic array run into the *aggregate update problem* (Hudak & Bloss, 1985). They need linear time to create a copy of the whole array, whereas imperative

languages accomplish the same task in constant time by destructively writing into the argument array.

Attempts to avoid or at least to mitigate this effect come at the expense of a more complex representation of arrays, which creates additional overhead elsewhere. For example, arrays could be represented as trees rather than as contiguous pieces of memory. Alternatively, "update" operations could merely be annotated internally, instead of actually being performed. While copy overhead is reduced, other array operations, e.g. element selection, become more expensive. Whether such measures improve performance or not depends on the mix of operations, but the conceptual problem remains.

Investigations involving CLEAN and HASKELL have shown that arrays need to be strict and unboxed in order to achieve acceptable runtime performance (van Groningen, 1997; Serrarens, 1997; Zörner, 1998; Chakravarty & Keller, 2003). Furthermore, array processing must be organized in a single-threaded manner, based on uniqueness types (Barendsen & Smetsers, 1995) or on state monads (Wadler, 1992). An alternative approach is taken by ID and ML. They introduce arrays as non-functional, stateful data structures. In all cases, most elegance of functional programming is lost. Arrays are allocated, copied, and removed under explicit control of the programmer.

Few functional languages have been designed with arrays in mind. NESL (Blelloch, 1996) supports nested vectors of varying length. These *sequences* are half way in between lists and multi-dimensional homogeneous arrays. They are particularly useful for processing irregularly structured numerical data. A NESL-specific optimization technique named *flattening* (Blelloch & Sabot, 1990) aims at increasing regularity in intermediate code in order to exploit the specific capabilities of vector processors. Investigations on runtime performance in multiprocessor environments have shown good results for applications that benefit from NESL's support for irregularity. For example, sparse matrix operations in NESL outperform equivalent dense implementations in FORTRAN (Blelloch *et al.*, 1994). However, if data is more regularly structured and the specific capabilities of NESL cannot be exploited, performance turns out to be inferior to FORTRAN.

SISAL (McGraw *et al.*, 1985) first showed that efficient processing of regular arrays is feasible in a functional setting, provided that both language design and implementation are tailor-made. In order to reduce runtime overhead, SISAL dispenses with many typical features of functional languages. SISAL neither supports partial applications nor higher-order functions or polymorphism; the evaluation order is strict. Memory management for arrays is based on reference counting rather than on garbage collection. Hence, the exact number of active references to an array is available at runtime. Array operations can update argument arrays that are not referenced elsewhere destructively without penetrating the functional semantics (Cann, 1989). Code optimizations that exploit referential transparency (Skedzielewski & Welcome, 1985; Cann & Evripidou, 1995) and implicit generation of parallel code (Haines & Böhm, 1993) have achieved competitive performance figures for numerical benchmarks in multiprocessor environments (Oldehoeft *et al.*, 1986; Cann, 1992).

Unfortunately, the level of abstraction in SISAL programming hardly exceeds that of imperative languages with specific array support, e.g. FORTRAN-95 (Adams *et al*., 1997) or ZPL (Chamberlain *et al*., 1998). All array operations must explicitly be specified by means of FOR-loops, a SISAL-specific array comprehension. The language neither provides built-in aggregate operations nor means to define such general abstractions. Moreover, SISAL only supports vectors. Multi-dimensional arrays must be represented as nested vectors of equal length. This limitation introduces additional indirections to array references, which incur increasing overhead with growing array rank (Scholz, 1997; Bernecky, 1997). These shortcomings are addressed in more recent versions, e.g. SISAL 2.0 (Oldehoeft, 1992) and SISAL-90 (Feo *et al*., 1995), but none of them have been implemented.

SAC (Single Assignment C) is a more recent functional array language (Scholz, 2003). Like SISAL, the language design of SAC aims at high runtime performance. Still, the level of abstraction in array processing considerably exceeds that of SISAL. The core syntax of SAC is a subset of C with a strict, purely functional semantics based on context-free substitution of expressions. Nevertheless, the meaning of functional SAC code coincides with the state-based semantics of literally identical C code. This design is meant to facilitate conversion to SAC for programmers with a background in imperative languages.

The language kernel of SAC is extended by multi-dimensional, stateless arrays. In contrast to other array languages, SAC provides only a very small set of built-in operations on arrays, mostly primitives to retrieve data pertaining to the structure and contents of arrays, e.g. rank, shape, or single elements. All aggregate array operations are specified in SAC itself using a versatile and powerful array comprehension construct, named *with-loop*. In contrast to the FOR-loops of SISAL, WITH-loops allow code to abstract not only from concrete shapes of argument arrays, but even from concrete ranks. Moreover, such rank-invariant specifications can be embedded within functions, which are applicable to arrays of any rank and shape.

By these means, most built-in operations known from FORTRAN-95 or from interpreted array languages like APL, J, or NIAL can be implemented in SAC itself without loss of generality (Grelck & Scholz, 1999). SAC provides a comprehensive selection of array operations in the standard library. Among others, the array module includes element-wise extensions of the usual arithmetic and relational operators, typical reduction operations like sum and product, various subarray selection facilities, as well as shift and rotate operations. In contrast to array support which is hard-wired into the compiler, our library-based solution is easier to maintain, to extend, and to customize for varying requirements.

SAC propagates a programming methodology based on the principles of abstraction and composition. Like in APL, complex array operations and entire application programs are constructed by composition of simpler and more general operations in multiple layers of abstractions. Unlike APL, the most basic building blocks of this hierarchy of abstractions are implemented by WITH-loops, not built-in. Whenever a basic operation is found to be missing during program development, it can easily be added to the repertoire and reused in future projects.

Despite the high-level APL-like programming style, SAC code has managed to outperform equivalent SISAL programs (Scholz, 1999). In several case studies SAC has also proved to be competitive even with low-level, machine-oriented languages (Grelck & Scholz, 2000; Grelck, 2002; Grelck & Scholz, 2003b; Scholz, 2003; Grelck & Scholz, 2003a). We achieve this runtime behaviour by the consequent application of standard compiler optimizations in conjunction with a number of tailor-made array optimizations. They restructure code from a representation amenable to programmers and maintenance towards a representation suitable for efficient execution by machines (Scholz, 1998; Scholz, 2003; Grelck & Scholz, 2003a; Grelck *et al.*, 2004). Both the general and the array-specific optimizations substantially benefit from the functional, side-effect free semantics of SAC.

The tailor-made language design makes SAC a suitable candidate for compiler-directed parallelization. Competitive sequential performance provides the opportunity to achieve real performance gains with respect to existing imperative implementations, not only in comparison with SAC uniprocessor runtimes. The particular challenge of high sequential performance is that realization of satisfying speedups becomes substantially more difficult. Fast sequential code is bad for the ratio between time spent in synchronization/communication operations and time spent in productive code. This ratio eventually determines parallel performance. With high sequential performance even minor inefficiencies in the organization of parallel program execution substantially degrade scalability. In this context, we have developed compilation techniques and runtime systems for the efficient parallel execution of high-level SAC array processing code on shared memory multiprocessors (Grelc, 1999, 2001, 2003). Our runtime system is based on the multi-threading standard PTHREADS (Institute of Electrical and Electronic Engineers, Inc., 1995).

We have chosen shared memory multiprocessors as compilation target for several reasons. This increasingly popular class of machines is in wide-spread use as small- and medium-range servers. The traditional scalability bottleneck of the shared memory is mitigated by processor-specific hierarchies of large and fast caches. Larger systems actually reduce the shared memory to a shared address space on top of physically distributed memory and a highly integrated interconnection network (Hennessy & Patterson, 2003). The Top500 list of the most powerful computing sites worldwide (Dongarra *et al.*, 2003) demonstrates the suitability of shared memory systems for complex computational tasks in practice.

Since SAC is a dedicated array language, typical SAC code is dominated by array operations. In one way or another all aggregate array operations in SAC are implemented by WITH-loops. Hence, WITH-loops form the ideal basis for parallelization, following a data parallel approach. Targeting shared memory architectures in general and PTHREADS in particular makes explicit decomposition of arrays and data exchange via message passing superfluous. Hence, integral parts of the existing sequential compiler framework can be reused with only minor alterations. Therefore, we can concentrate all effort on those aspects of the compilation process that are specific to multi-threaded execution.

Our initial compilation scheme focuses on the efficient parallel execution of individual WITH-loops. This leads to a *fork-join* execution model. A *master thread*

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

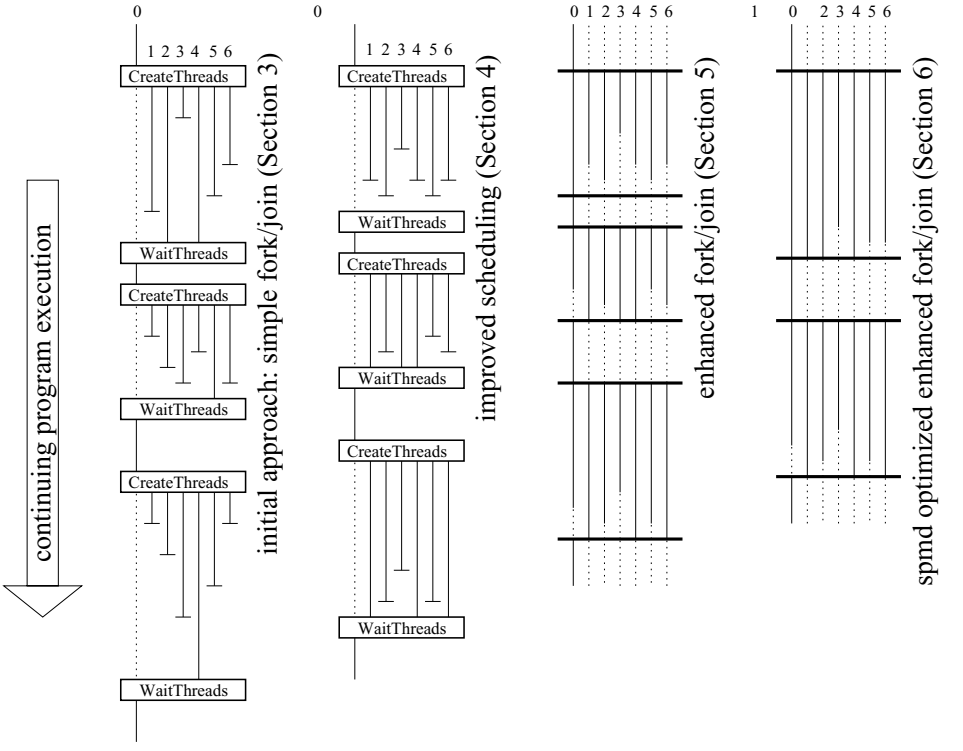

Fig. 1. Illustration of the refinement of compilation schemes and multi-threaded execution models from the initial approach (left) to the final solution (right).

executes a program just as in the sequential case. As soon as it reaches a WITH-loop, the master thread creates a fixed number of *worker threads*. The worker threads cooperatively compute the WITH-loop and terminate. Meanwhile, the master thread waits for the last worker thread to complete its share of work and resumes sequential execution thereafter.

The organization of multi-threaded program execution inevitably incurs some overhead. In order to achieve high parallel performance and reasonable scalability we must address all sources of overhead and try to avoid or to reduce them as far as possible. In our initial approach we can identify various sources of overhead: creation and termination of threads or synchronization and communication between threads. Moreover, program execution frequently stalls at barriers. These sources of overhead are addressed by a step-wise refinement of the initial compilation scheme and execution model. Figure 1 illustrates the refinement steps starting with the initial approach on the left hand side. Vertical lines represent threads computing over time. Dotted lines refer to sleeping threads. Horizontal boxes and bars represent synchronization and communication. The master thread has thread ID 0; the example uses 6 worker threads.

Our initial approach is characterized by a sequence of parallel execution steps interrupted by sequential phases. As a first refinement step, we investigate various WITH-loop scheduling techniques. WITH-loop scheduling is the process of partitioning

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

WITH-loop among worker threads for execution. Different approaches are developed that aim at a smooth distribution of workload, without ignoring such important issues as scheduling overhead or data locality. In Figure 1, this is indicated by adjusting the length of lines which represent computing threads.

In a second step, the initial fork/join execution model is refined to an enhanced fork/join model. Worker threads remain alive during the whole program execution. Costly thread creation and thread termination operations are replaced by tailor-made synchronization barriers. They aim at reducing the runtime overhead directly inflicted by synchronization and communication. In Figure 1, this is indicated by replacing horizontal boxes by thin bars. Moreover, the master thread participates in the cooperative execution of WITH-loops by temporarily turning itself into a worker thread rather than sleeping until completion of the WITH-loop.

In the third refinement step, regions of parallel activity are decoupled from individual WITH-loops. We introduce explicit skeleton-like intermediate language constructs for the internal representation of synchronization and communication patterns. Tailor-made optimization schemes aim at fusing subsequent steps of parallel execution into single ones. On the right hand side of Figure 1, this is illustrated by fusing the first and the second parallel section. This step reduces the number of synchronization barriers and creates larger regions of parallel execution, which also render scheduling techniques for workload distribution more effective.

The three refinement steps are presented in the order they have been developed. In fact, they are mostly independent of each other. Since they address different sources of overhead in different ways, they could be applied in any order. Still, all three refinement steps substantially benefit from each other. Only in conjunction they achieve the desired runtime performance in parallel execution.

The rest of the paper gives a comprehensive and detailed presentation and evaluation of the compilation techniques sketched out above. Section 2 introduces the array concept of SAC in general and the WITH-loop in particular. Section 3 elaborates on the compilation of individual WITH-loops into multi-threaded code. Various WITH-loop scheduling techniques are discussed in section 4. Section 5 refines the initial execution model towards the enhanced fork/join model. Section 6 describes the introduction of skeleton-like structures for the internal representation of synchronization and communication patterns and the fusion of multiple regions of parallel execution. Section 7 summarizes various experiments that investigate the suitability of the approach in general and the effectiveness of individual measures in particular. Section 8 draws conclusions.

2 Arrays and array operations

Arrays in SAC are represented by two vectors. The *shape vector* specifies an array's rank (number of axes or number of dimensions) and the number of elements along each axis. The *data vector* contains all elements of an array in row-major order. Their relationship is defined as follows. Let A be an n-dimensional array represented by the shape vector $\vec{s}_A = [s_0, \ldots, s_{n-1}]$ and by the data vector $\vec{d}_A = [d_0, \ldots, d_{m-1}]$.

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

Then the equation

$$m = \prod_{i=0}^{n-1} s_i$$

describes the correspondence between the shape vector and the length of the data vector. Moreover, the set of legal index vectors of the array A is defined as

$$\mathcal{IV}_A := \{[iv_0, \ldots, iv_{n-1}] \mid \forall j \in \{0, \ldots, n-1\} : 0 \leqslant iv_j < s_j\}.$$

An index vector $\vec{iv} = [iv_0, \ldots, iv_{n-1}]$ denotes the element d_k of the data vector $\vec{d}_A$ of array A if $\vec{iv}$ is a legal index vector of A, i.e. $\vec{iv} \in \mathcal{IV}_A$, and the equation

$$k = \sum_{i=0}^{n-1} \left(iv_i \ * \prod_{j=i+1}^{n-1} s_j \right)$$

holds. For example, the matrix

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 \\ 30 & 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\ 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 \end{pmatrix}$$

is represented by the shape vector `[5,10]` and the data vector

$$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, \ldots, 47, 48, 49].$$

The set of legal index vectors is

$$\{\, [i, j] \mid [0, 0] \leqslant [i, j] < [5, 10] \,\},$$

and the index vector `[3,2]` refers to element position 32 of the data vector.

As pointed out in the introduction, SaC provides only a very small set of built-in operations on arrays, mostly primitives to retrieve data pertaining to the structure and contents of arrays. For example, `shape(a)` and `dim(a)` yield the shape and the rank of an array `a`, respectively. Furthermore, `sel(iv,a)` yields the element of `a` at index position `iv`, provided that the length of the integer vector `iv` matches the rank of the array `a`. Applications of the form `sel(iv,a)` may be replaced by the more familiar notation `a[iv]`.

With-loops

All aggregate array operations are specified in SaC itself using a versatile and powerful array comprehension construct, named *with-loop*. With-loops are versatile language constructs for the specification of potentially complex `map`- and `fold`-like aggregate array operations. A definition of their syntax is given in Figure 2. Following the key word `with`, a with-loop is made up of a non-empty sequence of *parts* and a single *operation*. The operation determines the overall meaning of the with-loop. There are two variants: `genarray` and `fold`. With `genarray(Expr)` the with-loop creates a new array. The expression `Expr` must evaluate to an integer

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

```
WithLoopExpr  ⇒  with ⌈ Part ⌉+ Operation
Part          ⇒  ( Generator ) : Expr
Generator     ⇒  Expr <= Identifier < Expr ⌈ Filter ⌉
Filter        ⇒  step Expr ⌈ width Expr ⌉
Operation     ⇒  genarray ( Expr )
              |  fold ( FoldOp , Expr )
```

Fig. 2. Syntax definition of WITH-loops.

vector, which defines the shape of the array to be created. With `fold(` *`FoldOp`*`,` *`Expr`*`)` the WITH-loop specifies a reduction operation. In this case, *`FoldOp`* must be the name of an appropriate associative and commutative binary operation. The expression *`Expr`* must evaluate to the neutral element of *`FoldOp`*.

Each part consists of a *generator* and an expression. The expression is said to be *associated* with the generator. The generator defines a set of index vectors along with an *index variable* representing elements of this set. Two expressions, which must evaluate to non-negative integer vectors of equal length, define lower and upper bounds of a rectangular range of index vectors. For each element of the index vector set defined by a generator, the associated expression is evaluated. Depending on the operational variant of the WITH-loop, the value of the associated expression either defines the element value at the corresponding index position of the new array (`genarray`), or it is given as an argument to the fold operation (`fold`).

For example, the matrix

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 & 19 \\ 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 & 29 \\ 30 & 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\ 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 & 49 \end{pmatrix}$$

may be specified by the following WITH-loop:

```
with
  ([0,0] <= iv < [5,10]) : iv[0] * 10 + iv[1]
genarray( [5,10])
```

Similarly,

```
with
  ([0,0] <= iv < [5,10]) : iv[0] * 10 + iv[1]
fold( +, 0)
```

computes the sum of all elements of the example matrix without actually creating the matrix.

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

With multiple parts, disjoint index subsets of an array may be computed and initialized according to different specifications. For example,

```
with
  ([0,0] <= iv < [5, 8]) : iv[0] * 10 + iv[1]
  ([0,8] <= iv < [5,10]) : 0
genarray( [5,10])
```

yields

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 0 & 0 \\ 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 & 0 & 0 \\ 20 & 21 & 22 & 23 & 24 & 25 & 26 & 27 & 0 & 0 \\ 30 & 31 & 32 & 33 & 34 & 35 & 36 & 37 & 0 & 0 \\ 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 & 0 & 0 \end{pmatrix}$$

An optional filter may be used to restrict index sets to grids. For example,

```
with
  ([0,0] <= iv < [5,10] step [1,2]) : iv[0] * 10 + iv[1]
  ([0,1] <= iv < [5,10] step [1,2]) : 0
genarray( [5,10])
```

yields

$$\begin{pmatrix} 0 & 0 & 2 & 0 & 4 & 0 & 6 & 0 & 8 & 0 \\ 10 & 0 & 12 & 0 & 14 & 0 & 16 & 0 & 18 & 0 \\ 20 & 0 & 22 & 0 & 24 & 0 & 26 & 0 & 28 & 0 \\ 30 & 0 & 32 & 0 & 34 & 0 & 36 & 0 & 38 & 0 \\ 40 & 0 & 42 & 0 & 44 & 0 & 46 & 0 & 48 & 0 \end{pmatrix}$$

An additional *width* specification allows generators to define generalized grids as in the following example where

```
with
  ([0,0] <= iv < [5,10] step [4,4] width [2,2]) : 9
  ([0,2] <= iv < [5,10] step [4,4] width [2,2]) : 0
  ([2,0] <= iv < [5,10] step [4,1] width [2,1]) : 1
genarray( [5,10])
```

yields

$$\begin{pmatrix} 9 & 9 & 0 & 0 & 9 & 9 & 0 & 0 & 9 & 9 \\ 9 & 9 & 0 & 0 & 9 & 9 & 0 & 0 & 9 & 9 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 9 & 9 & 0 & 0 & 9 & 9 & 0 & 0 & 9 & 9 \end{pmatrix}$$

Expressions that define step and width vectors must evaluate to positive integer vectors of the same length as the bound vectors of the generator. The full range of generators can also be used with `fold`-WITH-loops.

To give a formal definition of index sets, let *a*, *b*, *s*, and *w* denote expressions that evaluate to appropriate vectors of length *n*. Then the generator

(*a* <= iv < *b* step *s* width *w*)

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

defines the following set of index vectors:

$$\{\ iv \mid \forall_{j\in\{0,\ldots,n-1\}} : a_j \leqslant iv_j < b_j \ \wedge\ (iv_j - a_j) \mod s_j < w_j\ \}.$$

Two constraints which cannot be expressed by means of syntax apply to WITH-loops. Firstly, the index sets defined by multiple generators must be pairwise disjoint. Secondly, for `genarray`-WITH-loops the union of all index sets must equal the set of legal index vectors of the result array, i.e. each element of the result array must be associated with exactly one generator.

An important property of WITH-loops is that evaluation of the associated expression for any element of the union of index sets is completely independent of all others, leaving a language implementation free to choose any suitable evaluation order. We exploit this property both for optimization and for parallelization. Consequently, WITH-loops form the basis for large-scale code restructuring optimizations, which combine several individual WITH-loops into a single one. Therefore, the average computational workload per array element is much higher than for the mostly light-weight built-in aggregate operations in other languages. Since increased workload per element generally improves the ratio between productive computations and runtime overhead, WITH-loops are particularly amenable to parallel execution.

Unfortunately, the various benefits of WITH-loops do not come for free. Whereas built-in array operations in other languages are almost always homogeneous, WITH-loops may represent considerably inhomogeneous operations. A single WITH-loop may consist of many generators defining disjoint, but potentially interleaved, index vector sets. Moreover, associated expressions may substantially differ from each other with respect to their computational complexity. Such inhomogeneities significantly complicate the generation of efficiently executable code.

In a sense, WITH-loops resemble algorithmic skeletons (Cole, 1989; Cole, 2004) liberated from restrictions imposed by their typical realization as higher-order functions in statically typed languages. Actually, `genarray`-WITH-loops are powerful array comprehensions, and `fold`-WITH-loops are versatile folding templates, both specifically designed for multi-dimensional arrays. By their semantics, they explicitly introduce fine-grained concurrency into SAC programs in a way that is particularly amenable to exploitation for parallel execution. Whether some WITH-loop actually is executed in parallel by multiple processors or sequentially by a single processor, is solely up to compiler and runtime system. Even within a single compiled code, the same WITH-loop may be executed in parallel sometimes and sequentially otherwise, depending on the state of the execution machinery. WITH-loops do not specify a pattern of parallel behaviour, but – in conjunction with the entire language design of SAC – they facilitate automatic or implicit parallelization.

3 Generating multi-threaded code

When contemplating the multi-threaded execution of SAC programs, it turns out that some program parts require sequential execution by a dedicated thread. For example, I/O operations must be performed in a single-threaded way in order to preserve the sequential I/O behaviour. I/O operations and state modifications in

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

general are properly integrated into the purely functional world of SaC by a variant of uniqueness types (Achten & Plasmeijer, 1995), named *classes* (Grelck & Scholz, 1995). Enforcing the uniqueness property bans all state modifications from bodies of WITH-loops. Hence, the easiest way to preserve sequential I/O behaviour is to restrict parallel program execution to individual WITH-loops.

Following this idea, a *master thread* executes most parts of a SaC program in a single-threaded way, similar to sequential execution. Only when it comes to computing a WITH-loop, the master thread creates the desired number of additional *worker threads*. Subsequently, all worker threads concurrently, but cooperatively, execute the single WITH-loop. Meanwhile, the master thread merely awaits the termination of the worker threads. After all worker threads have finished their individual shares of work, they terminate, and the master thread resumes sequential execution.

The PTHREADS standard does not provide any means to associate threads with processors. The host machine's operating system is responsible for scheduling worker threads to different processors in a reasonable way. In general, it is recommended to use exactly one thread per processor. This choice avoids frequent context switches between cooperating threads and proved optimal in extensive experiments.

In the following, we present detailed schemes for compiling individual WITH-loops into multi-threaded code. Our initial focus is on `genarray`-WITH-loops; `fold`-WITH-loops are covered towards the end of this section. Rules of the form

$$\mathscr{C}[\![expr]\!] \; = expr'$$

denote the context-free replacement of a SaC program fragment *expr* by a pseudo C program fragment *expr′*.

Compilation of `genarray`*-with-loops*

Figure 3 shows the compilation scheme $\mathscr{CMT}$ for generating multi-threaded C code from a single `genarray`-WITH-loop. Actually, two different fragments of C code are generated: one for the master thread and one for the worker threads. Their interaction is indicated by horizontal arrows.

Whenever execution of the master thread reaches a `genarray`-WITH-loop, it first allocates memory appropriate for storing the result array. Here and in the sequel, we assume that nested expressions have already been flattened by a pre-processing step. Hence, *shp* denotes a variable referring to the shape of the array to be created rather than a complex expression. Due to the commitment to shared memory architectures, explicit decomposition of arrays is obsolete. Hence, implicit dynamic memory management for arrays can be adopted from the existing sequential implementation with only minor adaptations. Arrays are still stored in a single contiguous address space. Memory is allocated by the master thread during single-threaded execution.

For worker threads to be able to compute their individual shares of a WITH-loop, it is necessary to set up an execution environment equivalent to that of the master thread. More precisely, all those variables referred to within the body of the WITH-loop, but defined before, must also exist in the worker thread. Moreover,

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

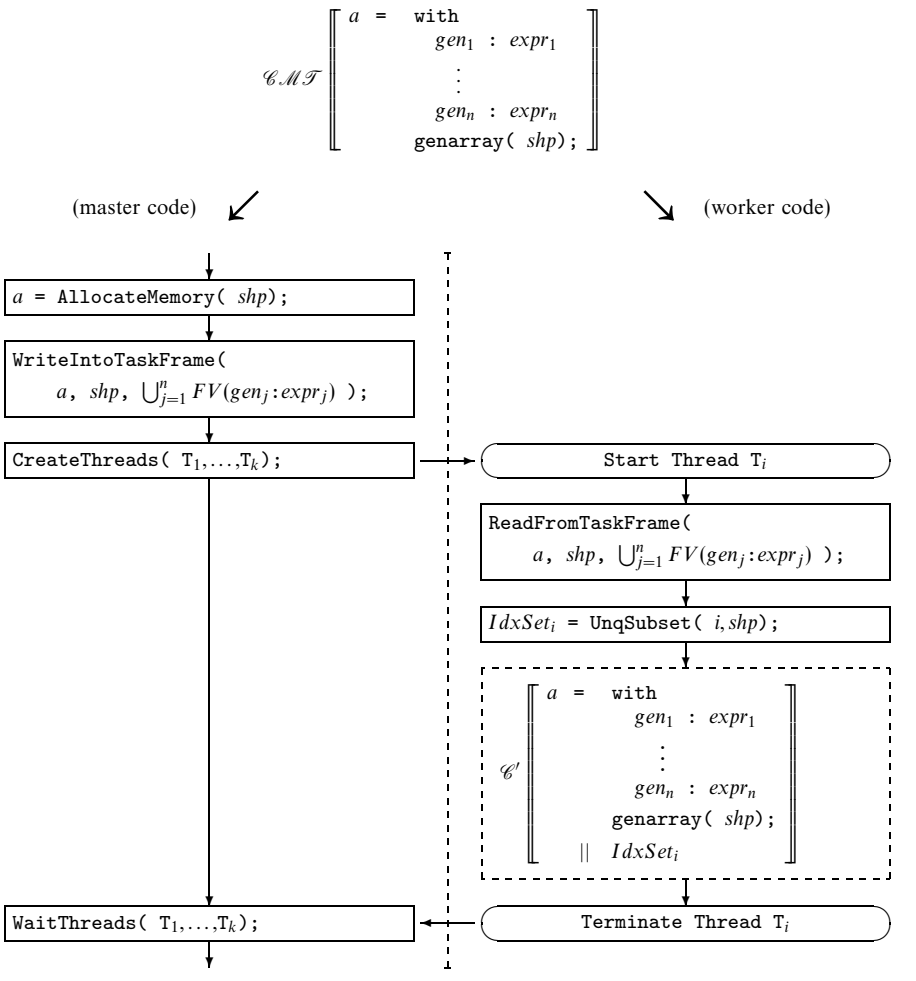

Fig. 3. Multi-threaded compilation scheme for `genarray`-WITH-loops.

they must be initialized to the current values in the master thread to ensure proper execution of the WITH-loop by each worker thread. The required information must be communicated to the worker threads before they start execution of productive code. In a shared memory environment communication is realized by writing to and reading from global data structures. Therefore, each WITH-loop in a SAC program is associated with a global buffer, called *task frame*. The task frame is tailor-made for communication between master thread and worker threads for that individual WITH-loop. In the case of a `genarray`-WITH-loop, the master thread – in addition to the values of all free variables of the various generator-expression pairs – stores the pre-allocated result array and its shape in the task frame. Due to the shared memory, it suffices to communicate the base addresses and the shapes of arrays, rather than the arrays themselves.

Eventually, the desired number of worker threads is created. The actual number may either be fixed at compile time, or it is determined by a command line parameter

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

or by an environment variable upon program start. In any case, the number of threads remains constant throughout an entire program run.

All worker threads uniformly execute the code shown on the right hand side of Figure 3, but each thread may identify itself by means of a unique ID. As a first step, a worker thread sets up its execution environment by extracting the required information from the task frame. Access to the task frame can always be granted without costly synchronization of threads via critical regions or similar. As long as the master thread writes to the task frame, it is known to be the only thread in the process. As soon as multiple threads exist, the task frame is used as a read-only buffer.

Although all worker threads execute exactly the same code, they must compute and initialize pairwise disjoint sets of result array elements for parallel execution to make sense. This additional side condition must be taken into account when eventually compiling a WITH-loop into nestings of FOR-loops in C. Even in the sequential case, generation of efficient code from WITH-loops with multiple interleaved generators has proved to be an extremely complicated and challenging task (Grelck *et al*., 2000; Scholz, 2003). Therefore, we want to reuse the existing sequential code generation scheme for WITH-loops as far as possible. However, we need to associate each element of the result array with exactly one thread. The *scheduling* of array elements to threads directly affects workload distribution among processors and is vital for performance.

The solution adopted in Figure 3 keeps code generation and scheduling as separate as possible. By means of the runtime system function `UnqSubset` each worker thread determines a rectangular index subspace $IdxSet_i$ based on its unique ID and the shape of the result array. Proper implementations of `UnqSubset` guarantee that each legal index position of the result array is covered by exactly one such index subspace. Different implementations of `UnqSubset` allow us to realize various different scheduling techniques without affecting the compilation scheme otherwise. Based on programmer annotations or on compiler heuristics, the most appropriate scheduler implementation may be selected. Section 4 further elaborates on this topic.

After initialization of the execution environment and allocation of a rectangular index subspace by the loop scheduler, execution of the WITH-loop by an individual worker thread proceeds almost as in the sequential case. In Figure 3, this is indicated by the recursive application of the code generation scheme $\mathscr{C}'$, which is a variant of the sequential code generation scheme $\mathscr{C}$ (Scholz, 2003). The most significant difference between $\mathscr{C}'$ and $\mathscr{C}$ is that the range of any FOR-loop in compiled code is restricted to the intersection of its original range and the rectangular index subspace $IdxSet_i$ allocated by the WITH-loop scheduler. After having completed its individual share of work, a worker thread terminates. The master thread waits for the longest-running worker thread to terminate and resumes sequential execution.

Code generation for worker threads

The adapted code generation scheme $\mathscr{C}'$ for worker threads is formally defined in Figure 4. The code starts by allocating memory to accommodate the index

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

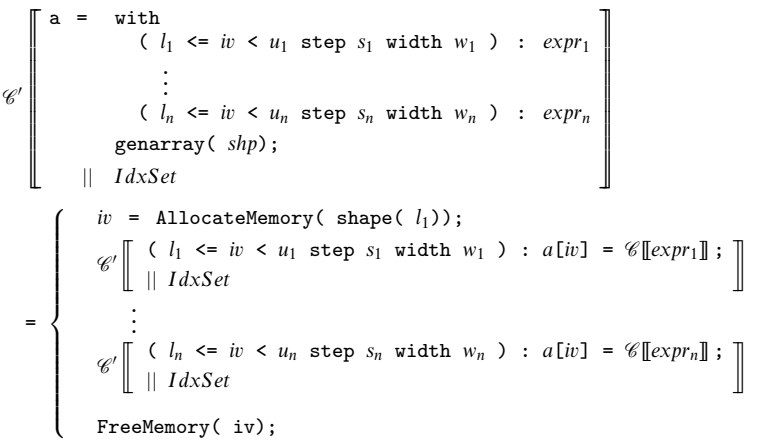

Fig. 4. Code generation scheme for `genarray`-WITH-loops.

variable `iv`. Afterwards, the compilation scheme $\mathscr{C}'$ is recursively applied to each part of the multi-generator WITH-loop individually. The specification of the local index subspace *IdxSet* is propagated as is. However, this is not done before the corresponding expressions to be evaluated are transformed into assignments of the form `a[iv] =` $\mathscr{C}[\![expr]\!]$, which ensures correct insertion of the computed values into the result array. Using the compilation scheme $\mathscr{C}$ here instead of $\mathscr{C}'$ reflects the fact that compilation of these expressions is independent of the choice of sequential or multi-threaded program execution. Finally, we de-allocate the memory holding the index variable because the index variable's scope is always confined to a single WITH-loop.

Generation of C FOR-loops for individual parts of a WITH-loop is defined in Figure 5. The compilation scheme $\mathscr{C}'$ generates either one or two FOR-loops for each dimension or axis of the index space. If the corresponding element of the width vector is known to evaluate to one, the scheme generates a single FOR-loop only. It runs on the intersection between the range defined by lower and upper bound of the generator, $\vec{l}$ and $\vec{u}$, and the local index subspace, which itself is defined by the lower and upper bound vectors $\vec{lb}$ and $\vec{ub}$. The corresponding step vector component determines the FOR-loop's increment.

If the current component of the width vector differs from one or is unknown at compile time, we must create two nested FOR-loops: an outer loop for initialization and stepwise increment of the corresponding index variable component and an inner loop for realization of the width component. The body of the inner loop is derived from recursive application of the code generation scheme $\mathscr{C}'$ with the leading elements of the four generator vectors removed. The innermost loop body is defined by the pregenerated assignment, which is specific to the WITH-loop variant.

As pointed out before, one important property of WITH-loops is the freedom to choose any suitable evaluation order. We prominently exploit this property for compilation into multi-threaded code. In a less obvious way, the property is also

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

$$\mathscr{C}'\left[\!\!\left[\begin{array}{l}\texttt{([}l_i,\ldots,l_{m-1}\texttt{] <= } iv \texttt{ < [}u_i,\ldots,u_{m-1}\texttt{]}\\ \quad \texttt{step [}s_i,\ldots,s_{m-1}\texttt{] width [}1,w_{i+1},\ldots,w_{m-1}\texttt{])} : \mathit{Assign}\\ \quad \texttt{||} \quad \texttt{[}lb_i,\ldots,lb_{m-1}\texttt{]} \quad \texttt{[}ub_i,\ldots,ub_{m-1}\texttt{]}\end{array}\right]\!\!\right]$$

$$= \left\{\begin{array}{l}\texttt{for (} iv\texttt{[}i\texttt{] = max(} l_i\texttt{,} lb_i\texttt{); } iv\texttt{[}i\texttt{] < min(} u_i\texttt{,} ub_i\texttt{); } iv\texttt{[}i\texttt{] += } s_i\texttt{) \{}\\ \quad \mathscr{C}'\left[\!\!\left[\begin{array}{l}\texttt{([}l_{i+1},\ldots,l_{m-1}\texttt{] <= } iv \texttt{ < [}u_{i+1},\ldots,u_{m-1}\texttt{]}\\ \quad \texttt{step [}s_{i+1},\ldots,s_{m-1}\texttt{] width [}w_{i+1},\ldots,w_{m-1}\texttt{])} : \mathit{Assign}\\ \quad \texttt{||} \quad \texttt{[}lb_{i+1},\ldots,lb_{m-1}\texttt{]} \quad \texttt{[}ub_{i+1},\ldots,ub_{m-1}\texttt{]}\end{array}\right]\!\!\right]\\ \texttt{\}}\end{array}\right.$$

$$\mathscr{C}'\left[\!\!\left[\begin{array}{l}\texttt{([}l_i,\ldots,l_{m-1}\texttt{] <= } iv \texttt{ < [}u_i,\ldots,u_{m-1}\texttt{]}\\ \quad \texttt{step [}s_i,\ldots,s_{m-1}\texttt{] width [}w_i,\ldots,w_{m-1}\texttt{])} : \mathit{Assign}\\ \quad \texttt{||} \quad \texttt{[}lb_i,\ldots,lb_{m-1}\texttt{]} \quad \texttt{[}ub_i,\ldots,ub_{m-1}\texttt{]}\end{array}\right]\!\!\right]$$

$$= \left\{\begin{array}{l}\texttt{for (} iv\texttt{[}i\texttt{] = max(} l_i\texttt{,} lb_i\texttt{); } iv\texttt{[}i\texttt{] < min(} u_i\texttt{,} ub_i\texttt{); } iv\texttt{[}i\texttt{] += } s_i \texttt{ - } w_i\texttt{) \{}\\ \quad \texttt{stop = min(} iv\texttt{[}i\texttt{] + } w_i\texttt{, min(} u_i\texttt{,} ub_i\texttt{));}\\ \quad \texttt{for (; } iv\texttt{[}i\texttt{] < stop; } iv\texttt{[}i\texttt{] += 1) \{}\\ \quad\quad \mathscr{C}'\left[\!\!\left[\begin{array}{l}\texttt{([}l_{i+1},\ldots,l_{m-1}\texttt{] <= } iv \texttt{ < [}u_{i+1},\ldots,u_{m-1}\texttt{]}\\ \quad \texttt{step [}s_{i+1},\ldots,s_{m-1}\texttt{] width [}w_{i+1},\ldots,w_{m-1}\texttt{])} : \mathit{Assign}\\ \quad \texttt{||} \quad \texttt{[}lb_{i+1},\ldots,lb_{m-1}\texttt{]} \quad \texttt{[}ub_{i+1},\ldots,ub_{m-1}\texttt{]}\end{array}\right]\!\!\right]\\ \quad \texttt{\}}\\ \texttt{\}}\end{array}\right.$$

$$\mathscr{C}'\left[\!\!\left[\texttt{([] <= iv < [] step [] width []) :} \mathit{Assign} \quad \texttt{||} \quad \texttt{[] []}\right]\!\!\right] \quad = \quad \mathit{Assign}$$

Fig. 5. Code generation scheme for individual parts.

exploited by the code generation scheme $\mathscr{C}'$, as defined in Figure 4 and in Figure 5. Addressing each generator of a multi-generator WITH-loop in isolation substantially facilitates the compilation process. Unfortunately, this straightforward approach has two major disadvantages with respect to achieving high runtime performance.

- Separate compilation of generators introduces a considerable amount of loop overhead, in particular if step and width vectors are used.
- Cache memory utilization is poor because data transfer between main memory and cache memory in whole cache lines favours consecutive memory references to adjacent addresses (Hennessy & Patterson, 2003).

Although elaborate C compilers provide optimizations for rearranging loops (Zima & Chapman, 1991; Wolfe, 1995; Allen & Kennedy, 2001), they usually fail to improve compiled SAC code in a substantial way because they lack information about the special properties of the code. For example, C compilers do not *a priori* know that all innermost loop bodies can be computed independently of each other or that each result array element is initialized exactly once. Instead, this information must be inferred from the code, which proves to be quite difficult in the context of C. C compilers hardly achieve the required accuracy and must make conservative assumptions.

To overcome these limitations, the sequential code generation scheme $\mathscr{C}$ is extended by a post-processing phase, which systematically transforms generator-centric nestings of FOR-loops into equivalent ones that obey the *canonical order* (Grelck *et al.*, 2000). Elements of the result array are computed and initialized

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

in strictly ascending order according to the array's memory representation. The canonical order overcomes both deficiencies of the initial approach, excessive loop overhead and poor cache utilization. This post-processing phase has turned out to be crucial for overall performance. Fortunately, the design of the compilation scheme $\mathscr{C}\mathscr{M}\mathscr{T}$ allows us to reuse the sequential post-processing phase for worker thread code without modification. Therefore, we omit a detailed description here and refer to (Grelck *et al.*, 2000) and (Scholz, 2003) for additional information.

Compilation of `fold`*-with-loops*

Figure 6 shows the generation of multi-threaded code for `fold`-WITH-loops. Worker threads initialize a local accumulation variable $\mathtt{a}_i$ by the neutral element of the fold operation. Each worker thread computes a partial fold result based on a subset of the values to be folded. Upon termination of the worker threads, the master thread collects their partial fold results and computes the overall result. Unlike some notions of `fold` skeletons, e.g. as described in Michaerlson *et al.* (2001), parallel activity unfolds and resumes all at once rather than in a divide-and-conquer style.

Compiled code for the master thread starts by writing the values of the WITH-loop body's free variables into the corresponding task frame. The same happens to the fold operation's neutral element. Analogously to the result array shape *shp* in the compilation of `genarray`-WITH-loops, we assume *neutral* to refer to a simple variable. Having set up the task frame, the master thread creates the desired number of worker threads. Worker threads set up their execution environment, retrieved from the task frame. After that, each worker thread identifies a unique index subspace by calling the loop scheduler `UnqSubset`. However, the loop scheduler requires the shape of the overall index space as an argument. In contrast to `genarray`-WITH-loops, this information is not immediately available in `fold`-WITH-loops. Instead, it must be determined from the set of generators. In Figure 6, this is expressed by the pseudo operation `Closure`, which yields the smallest rectangular hull of the transitive closure of the generators.

Recursive application of the adapted code generation scheme $\mathscr{C}'$ to the WITH-loop itself and the description of the unique index subspace $IdxSet_i$ results in code that computes a partial fold result, which is stored in the (thread-) local accumulation variable $\mathtt{a}_i$. Upon completion, a worker thread writes its partial fold result into the task frame and terminates. In order to avoid costly synchronization upon concurrent access to the task frame by worker threads, the task frame provides a dedicated entry for each worker thread. The master thread awaits termination of all worker threads before it extracts the partial fold results from the task frame. Finally, the master thread itself combines the various partial fold results to generate the overall result and resumes sequential execution.

Figure 7 shows the definition of the adapted code generation scheme $\mathscr{C}'$ for `fold`-WITH-loops. First, a worker thread initializes a local accumulation variable `a` by the neutral element of the fold operation and allocates memory for the index variable `iv`. After that, the compilation scheme $\mathscr{C}'$ is recursively applied to each generator with the fold operation pregenerated. By abstracting from the actual operation to

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

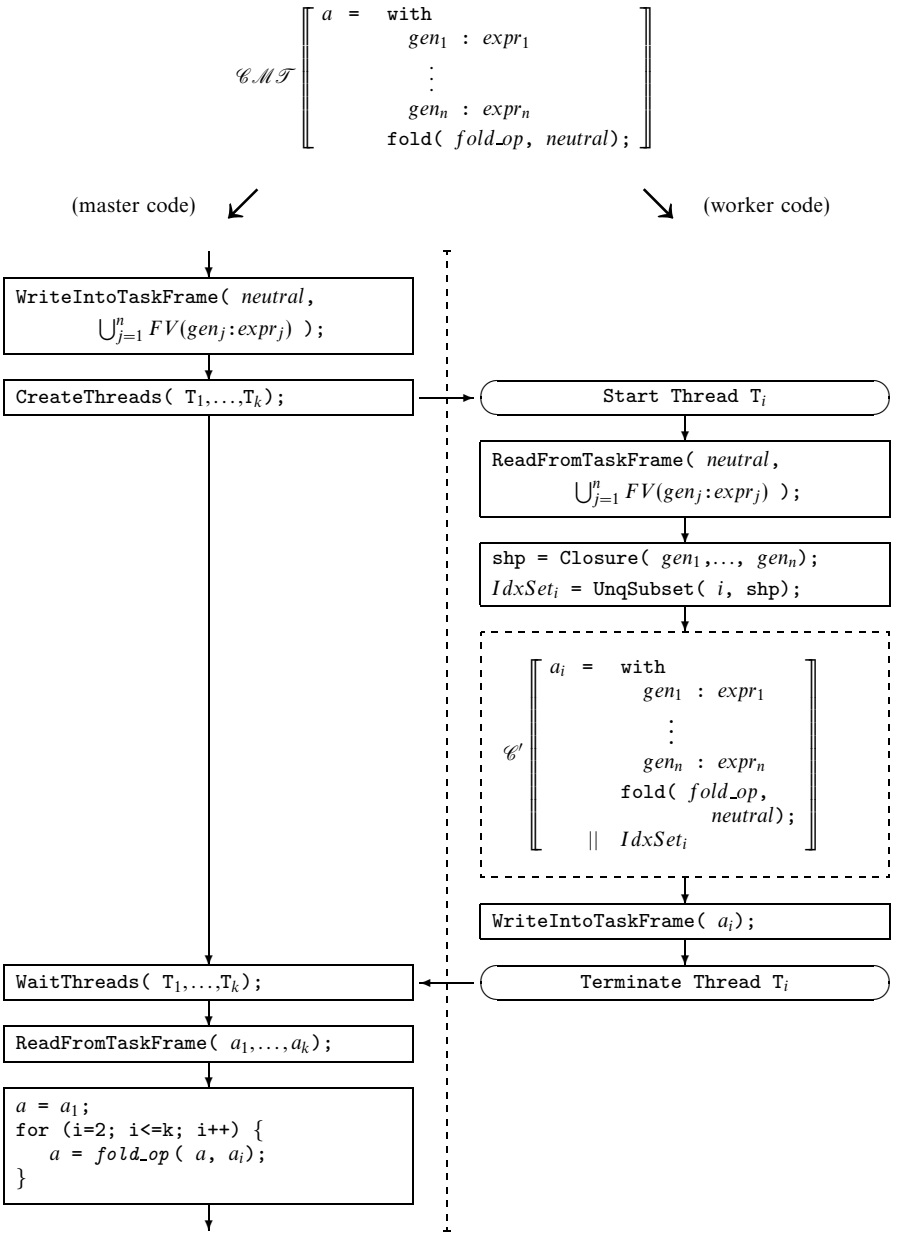

Fig. 6. Multi-threaded compilation scheme for `fold`-WITH-loops.

be inserted into the innermost FOR-loop body, the loop generation scheme defined in Figure 5 can be reused for `fold`-WITH-loops without modification.

Nested with-loops

Raising the focus from compilation intricacies of individual WITH-loops to the execution of entire programs, it becomes clear that program execution always

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

$$
\mathscr{C}'\left[\!\!\left[\begin{array}{l}
a = \texttt{with} \\
\quad \texttt{(}\ l_1 \texttt{ <= } iv \texttt{ < } u_1 \texttt{ step } s_1 \texttt{ width } w_1\ \texttt{)}\ \texttt{:}\ expr_1 \\
\quad\quad \vdots \\
\quad \texttt{(}\ l_n \texttt{ <= } iv \texttt{ < } u_n \texttt{ step } s_n \texttt{ width } w_n\ \texttt{)}\ \texttt{:}\ expr_n \\
\quad \texttt{fold(}\ fold\_op\texttt{,}\ neutral\texttt{);} \\
\quad \texttt{||}\ IdxSet
\end{array}\right]\!\!\right]
$$

$$
= \left\{\begin{array}{l}
a = neutral\texttt{;} \\
iv = \texttt{AllocateMemory(shape(}\ l_1\texttt{));} \\
\mathscr{C}'\left[\!\!\left[\begin{array}{l}\texttt{(}\ l_1 \texttt{ <= } iv \texttt{ < } u_1 \texttt{ step } s_1 \texttt{ width } w_1\ \texttt{)}\ \texttt{:}\ a = fold\_op(\ a, \mathscr{C}[\![expr_1]\!]\texttt{);} \\ \texttt{||}\ IdxSet\end{array}\right]\!\!\right] \\
\quad \vdots \\
\mathscr{C}'\left[\!\!\left[\begin{array}{l}\texttt{(}\ l_n \texttt{ <= } iv \texttt{ < } u_n \texttt{ step } s_n \texttt{ width } w_n\ \texttt{)}\ \texttt{:}\ a = fold\_op(\ a, \mathscr{C}[\![expr_n]\!]\texttt{);} \\ \texttt{||}\ IdxSet\end{array}\right]\!\!\right] \\
\texttt{FreeMemory(iv);}
\end{array}\right.
$$

Fig. 7. Code generation scheme for `fold`-WITH-loops.

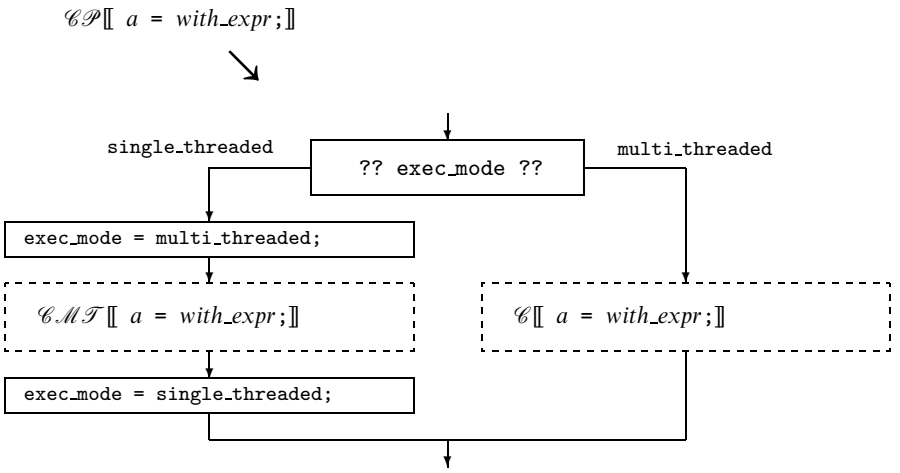

Fig. 8. Generalized compilation scheme $\mathscr{CP}$.

is in one of two states. Either WITH-loops are evaluated by a fixed number of worker threads in parallel, or some other code is executed sequentially by the master thread. However, WITH-loops may be nested, both statically in the code or dynamically in the unfolding of program execution. Program execution may already be in multi-threaded mode when it reaches a WITH-loop again. Since the number of threads already equals the number of available processors, we must prevent a recursive unfolding of parallel activity. WITH-loops that are nested within others, either statically or dynamically, must be executed sequentially. This is achieved by embedding the compilation scheme $\mathscr{CMT}$ within a more general scheme $\mathscr{CP}$, as shown in Figure 8.

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

The runtime system maintains a global flag `exec_mode`, which keeps track of the current state of program execution. Whenever program execution encounters a WITH-loop, the value of the `exec_mode` flag is inspected. Depending on its state, program execution either branches into multi-threaded code (generated by the compilation scheme $\mathscr{CMT}$) or into sequential code (generated by the compilation scheme $\mathscr{C}$). In the former case, the `exec_mode` flag is inverted during execution of the multi-threaded code. Access to the global state flag goes without protection because any write operation on it is guaranteed to be executed by the master thread while it is the only thread of the process. Inspection of the `exec_mode` flag can be avoided for statically nested WITH-loops. Moreover, simple heuristics identify WITH-loops for which parallel execution is likely not to pay off.

Although the compilation scheme $\mathscr{CP}$ restricts the unfolding of parallel activity to a single level, this does not mean sequential execution of all WITH-loops that are nested within others. A SAC-specific optimization technique named WITH-LOOP-SCALARIZATION (Grelck *et al.*, 2004) aims at transforming static nestings of WITH-loops into single ones – regardless of whether code is compiled for multi-threaded or for sequential execution. In any case, it is reasonable to assume that an outermost WITH-loop provides sufficiently fine-grained concurrency to exploit all available processors. This is an advantage of processing homogeneous arrays instead of lists or trees. Moreover, in shared memory environments processor counts are limited by the trade-off between scalability and hardware cost. Dispensing with nested levels of parallelism, the entire problem space discussed in the context of nested algorithmic skeletons (see, for example, Hamdan *et al.* (1999) and Michaelson *et al.* (2001)) can be avoided. As a consequence, our runtime system is significantly more efficient for realistic applications. This can be observed when comparing SAC performance figures with, for example, those in Hamdan *et al.* (1999) and Michaelson *et al.* (2001).

4 With-loop scheduling

WITH-loop scheduling is the process of allocating each element of the global index space defined by a WITH-loop to exactly one worker thread. This worker thread is then responsible for evaluation of the associated expression and for initialization of the corresponding element of the result array (`genarray`-WITH-loop) or for execution of the folding operation (`fold`-WITH-loop). Conceptually, each element of the global index set could be processed in parallel by a different thread. However, the useful number of threads is limited by the number of available processors. Typically, the size of the global index set by far exceeds the number of worker threads. Therefore, it is necessary to allocate entire index subspaces to worker threads rather than individual elements of the global index set. To facilitate compilation into efficiently executable code, the compilation schemes outlined in the previous section only support dense rectangular index subspaces. Similar to WITH-loop generators, they are represented by two boundary vectors.

In addition to these technical requirements, the main task of WITH-loop scheduling is to distribute the computational workload among threads as evenly as possible. Since the parallel execution of a WITH-loop is completed by a barrier synchronization,

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

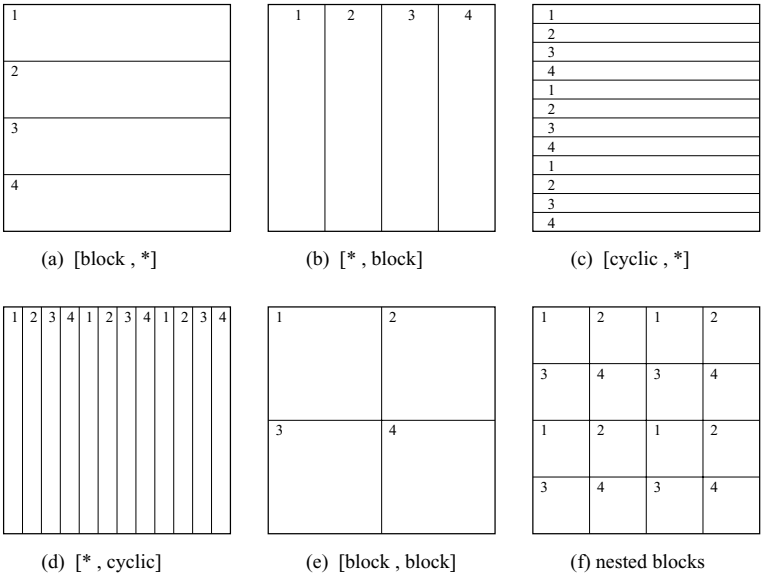

Fig. 9. Examples of static scheduling schemes.

overall execution time is determined by the "slowest" thread. Successful workload balancing is one major prerequisite for achieving good parallel performance. Unfortunately, the process of finding a suitable workload distribution contributes to organizational overhead. Hence, the scheduling problem is characterized by a trade-off between quality of workload balancing and overhead produced. Since the scheduling problem itself is not SAC-specific, we summarize existing approaches first and describe the solution adopted for the specific conditions of our compilation framework towards the end of the section.

Static scheduling

Two classes of loop scheduling techniques can be distinguished: *static* and *dynamic* approaches. Static techniques employ an a-priori association of loop iterations with threads. Well-known representatives of this class are *block* scheduling, *cyclic* scheduling, and *block-cyclic* scheduling. They are illustrated in Figure 9 for the 2-dimensional case and four threads. Block scheduling subdivides the iteration space into disjoint rectangular blocks, one for each thread. Cyclic scheduling associates every n-th iteration with the same thread. As a combination of both, block-cyclic scheduling subdivides the iteration space into blocks of a fixed size and applies cyclic scheduling on the level of these blocks. In the context of multi-dimensional loops or loop nestings, additional degrees of freedom arise from the selection of one or several scheduling dimensions and the possibility to combine different techniques along different axes of the iteration space.

Static scheduling techniques have been studied intensively in the context of HPF and its predecessors (Tseng, 1993; Loveman, 1993; Koebel *et al.*, 1994; Roth, 1997). If the computational workload is regularly distributed over the iteration space, block

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

scheduling combines low overhead with good workload balancing and high data locality. Cyclic scheduling is the choice whenever workload depends on the loop index, e.g. when processing triangular matrices. However, on modern cache-based computer architectures cyclic scheduling suffers from poor data locality. It is usually replaced by block-cyclic scheduling, which often is a reasonable compromise between data locality and workload balancing.

Dynamic scheduling

Multiprocessor systems rarely run a single application only. Threads of compiled SaC code compete for limited computing resources with unrelated processes incidentally running on the same system. As a matter of principle, static scheduling techniques cannot reflect changing workload characteristics of machines which run several independent processes. This scenario asks for *dynamic* or *adaptive* scheduling techniques.

Early examples such as *self-scheduling* (Tang & Yew, 1986) or *uniform-sized chunking* (Kruskal & Weiss, 1985) associate individual iterations or entire sets of iterations to threads on demand. A central task queue stores tasks ready to be computed. Each thread retrieves a new task from the task queue as soon as it has finished its previous task. This technique ensures proper balancing even in the presence of irregular workload distributions and changing availability of processors. Unfortunately, this advantage comes at the expense of considerable organizational overhead for managing the central task queue. Shared among all threads, the task queue constitutes a critical region and requires synchronization upon each access.

A key factor for successful application of dynamic scheduling techniques is the choice of a suitable granularity, i.e. the size of the tasks or groups of iterations as basis for the demand-driven allocation to threads. While a large number of small tasks results in good load balancing, the organizational overhead becomes prohibitive. Low granularity keeps the overhead within bounds, but load balancing may be insufficient. More advanced scheduling schemes address this dilemma by adapting task granularity during execution. They start with rather large tasks for low overhead in the beginning and systematically reduce task sizes for good load balancing towards approaching the final synchronization barrier. Examples are *guided self-scheduling* (Polychronopoulos & Kuck, 1987), *factoring* (Hummel *et al.*, 1992), or the *trapezoid method* (Tzen & Ni, 1993). Basically, they differ in the design of the task size regression scheme.

Another important source of overhead arises from the central task queue design. With increasing task granularity and growing processor counts, the central task queue is likely to become a performance bottleneck. Even worse, data locality is usually poor because schedulers based on a central task queue do not take into account the history of tasks previously associated with some thread (Squillante & Lazowska, 1993). Since modern shared memory multiprocessor systems heavily rely on effective utilization of processor-specific cache hierarchies, more recent dynamic scheduling techniques such as *locality-based dynamic scheduling (LDS)* (Li *et al.*, 1993), *affinity scheduling* (Markatos & LeBlanc, 1994), or *adaptive affinity scheduling*

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

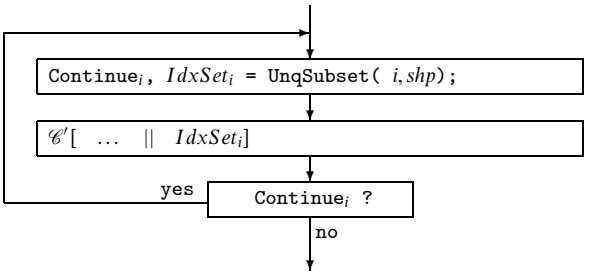

Fig. 10. Extending $\mathscr{CMT}$ for dynamic workload balancing.

(Yan *et al.*, 1997) operate on thread-specific task queues. Based on a static a-priori allocation of tasks to threads, under-utilized threads "steal" tasks from heavier loaded threads once they have completed their own *a priori* assignment.

Adapting compilation schemes for dynamic scheduling

A common characteristic of all dynamic loop scheduling techniques is the repeated allocation of tasks to individual threads. This is a prerequisite for any kind of adaptive workload balancing. Unfortunately, our multi-threaded compilation scheme $\mathscr{CMT}$, as defined in Section 3, associates exactly one rectangular index subspace with each worker thread. To support dynamic WITH-loop scheduling, we extend the compilation scheme $\mathscr{CMT}$ as illustrated in Figure 10. In addition to computing some iteration subspace, a scheduler provides a flag `Continue`, which decides whether or not the scheduler wishes to re-assign additional work to the thread later on. Once a worker thread has completed processing the assigned index subspace, it inspects the flag `Continue`. Depending on the value of `Continue`, the worker thread either continues as described in the previous section or it returns to the scheduler for assignment of further work.

Task selection and scheduling

Implementations of WITH-loop schedulers must address two more or less orthogonal aspects: task selection and task scheduling. Task selection combines individual elements of the global index set to tasks; task scheduling allocates entire tasks to threads. By choosing the number and size of tasks, task selection determines the granularity of task scheduling. Both task selection and task scheduling offer a wide range of design choices. Previous research on loop scheduling in general has shown that different settings require different scheduling techniques. No single approach is optimal in all cases. We have implemented two different task selectors and three different task schedulers. Their combinations provide the necessary flexibility to realize most of the loop scheduling techniques sketched out before.

The task selector `Even(`n`)` organizes the iteration space as equally sized rectangular blocks. The parameter n defines the desired number of tasks as a multiple of

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

Table 1. *Example application of task selector* `Factoring`

Task	Size	Task	Size	Task	Size	Task	Size
T1	101	T9	25	T17	6	T25	2
T2	101	T10	25	T18	6	T26	2
T3	101	T11	25	T19	6	T27	2
T4	101	T12	25	T20	6	T28	2
T5	50	T13	13	T21	3		
T6	50	T14	13	T22	3		
T7	50	T15	13	T23	3		
T8	50	T16	13	T24	3		

the number of threads. The task selector `Factoring` defines tasks with decreasing size according to a formula proposed in Hummel *et al.* (1992):

$$task\ size\ =\ \left\lfloor \frac{remaining\ iterations}{2 * number\ of\ threads} \right\rfloor\ +\ 1.$$

One task size is used for as many consecutive tasks as threads are used. The term *remaining iterations* refers to the number of iterations not yet allocated to tasks. Table 1 illustrates this task selector for an example of 800 iterations and four threads. In general, the factoring approach has shown to be a suitable compromise for the task granularity trade-off.

The three task schedulers are called `Static`, `Self`, and `Affinity`. The task scheduler `Static` offers the simplest implementation; it assigns tasks to threads statically in a round-robin manner. In combination with the task selector `Even(1)`, the task scheduler `Static` realizes a standard block scheduling. The general case `Even(`n`)` leads to a block-cyclic scheduling and, with increasing n, to cyclic scheduling. The task scheduler `Self` implements a central task queue, from which worker threads obtain tasks on demand. Depending on the choice of the task selector, the task scheduler `Self` either implements self-scheduling or factoring. The task scheduler `Affinity` requires the most complex implementation. It employs an individual task queue for each thread. As long as the associated local task queue is not empty, a thread always selects the next task pre-assigned to it. After having completed all statically pre-assigned tasks, a thread tries to identify heavily loaded threads with non-empty local task queues and "steals" tasks from them. Affinity scheduling offers all load balancing opportunities of a dynamic loop scheduler while avoiding the drawbacks of a central task queue design, i.e. frequent synchronization on access, low data locality, and the potential for becoming a scalability bottleneck.

For the time being, task selector `Even(1)` and task scheduler `Static` are used as defaults. However, for experimental purposes programmers are free to choose any combination of task selector and task scheduler either on a program-wide basis or for individual WITH-loops. Based on experimental experience, we envision compiler heuristics that implicitly select the most appropriate combination of task selector and task scheduler in the future.

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

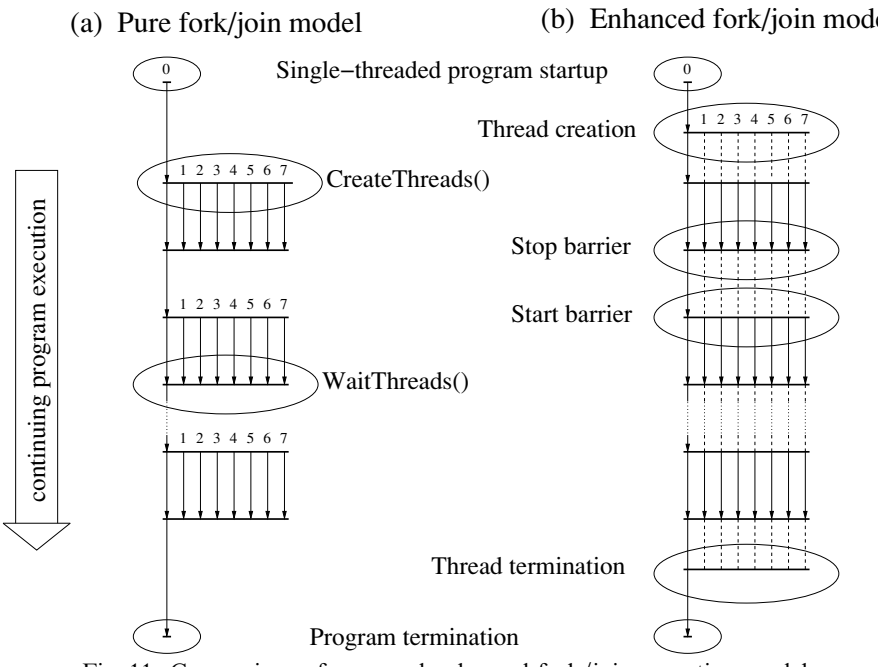

Fig. 11. Comparison of pure and enhanced fork/join execution models.

5 Enhancing the execution model

The compilation scheme $\mathscr{CMT}$, as defined in section 3, addresses individual WITH-loops. In compiled code, there is no connection between multi-threaded execution of consecutive WITH-loops. This limited scope facilitates the definition of a basic compilation scheme, yet it excludes any optimization across multiple WITH-loops. Following the multi-threaded execution of one WITH-loop, all worker threads terminate during synchronization. The same number of worker threads is again created for the multi-threaded execution of the following WITH-loop. Program execution is a sequence of steps alternatingly performed in single-threaded and in multi-threaded mode, as illustrated on the left hand side of Figure 11.

This fork/join execution model is conceptually simple. Synchronization and communication events are confined to thread creation and thread termination. Worker threads do not interact with each other at all. However, the price for simplicity is excessive runtime overhead due to frequent creation and termination of threads. Although the associated costs are much smaller than those for process creation and termination, they are still prohibitive for successful parallelization.

A solution which combines the conceptual simplicity of the fork/join approach with an efficient execution scheme is shown on the right hand side of Figure 11. In the *enhanced fork/join model*, the desired number of worker threads is created once at program start, and all threads remain active until the whole program terminates. Two tailor-made barriers, the *start barrier* and the *stop barrier*, realize all necessary synchronization among threads.

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

```
START_BARRIER_WAIT( )
{
  while (local_flag == global_flag);
  local_flag = global_flag;
}

START_BARRIER_LIFT( )
{
  global_flag = 1 - global_flag;
}
```

Fig. 12. Implementation of start barriers.

After creation, worker threads immediately hit a start barrier. As soon as the master thread encounters the first WITH-loop, the start barrier is lifted. The worker threads thereupon activated share the computation of the WITH-loop, exactly as in the pure fork/join model. Unlike in the pure fork/join model, the master thread temporarily turns itself into a worker thread and joins the other threads in the cooperative execution of the WITH-loop. Regular worker threads that have completed their individual computations pass the following stop barrier and, with nothing else to do, immediately move on to the next start barrier. After having finished its own assignment of work, the master thread waits at the stop barrier for the longest-running worker thread to arrive. Only then the master thread proceeds with subsequent (sequential) computations.

Implementing synchronization barriers

The combination of a stop barrier and a subsequent start barrier represents a full barrier synchronization, which is known to cause considerable runtime overhead with increasing numbers of threads (Cohen *et al.*, 1994; Hill & Skillicorn, 1998). Therefore, careful design and efficient implementation of start and stop barriers are crucial for performance. Figure 12 shows our implementation of the start barrier. It is based on a global flag, which is shared by all threads, and on one local flag within the scope of each worker thread. All flags are statically initialized to zero. Worker threads executing `START_BARRIER_WAIT` block on the condition of the empty WHILE-loop. By inverting the global flag during execution of `START_BARRIER_LIFT`, the master thread activates the worker threads. Having passed the barrier, worker threads copy the new value of the global flag into their local flags to prepare execution of subsequent instances of the start barrier.

Inverting the global flag while worker threads continuously poll on it, is a race condition. However, this race condition is without problems. On the one hand, only the master thread has write access to the global flag. On the other hand, worker threads only reflect the change of the flag's value, not the specific state of the memory location. By exploiting a controlled race condition this start barrier implementation completely avoids expensive thread synchronization mechanisms such as mutex locks or semaphores. The sole assumption made on the memory consistency model is that write operations issued by one processor are observed by others.

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

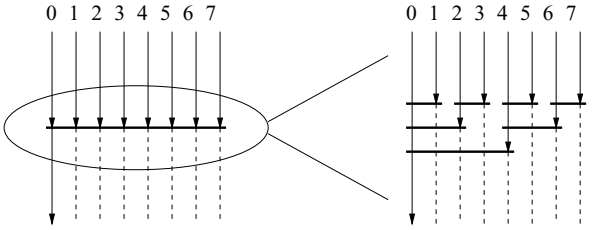

Fig. 13. Organization of tree-structured stop barriers.

One may argue that iteratively reading the global flag in very short time intervals generates heavy contention on the shared memory. In fact, the opposite is true for modern multiprocessors with processor-specific cache memories and hardware cache coherence. As soon as an individual worker thread hits the start barrier, it loads the global flag into the local cache of the processor it is currently running on. Afterwards, it only accesses the local cache when polling on the global flag. However, when the master thread inverts the global flag and writes its new value back to memory, the cache coherence mechanism invalidates the local copies in all other caches. Only then worker threads reload the global flag from memory and proceed beyond the start barrier. Each thread performs at most two main memory accesses during execution of the start barrier.

Our stop barrier implementation follows the same approach as the start barrier. It employs a global vector of flags, one for each thread. To improve scalability, the stop barrier is implemented as a binary tree of pairwise synchronizations, as illustrated in Figure 13. After having set its associated ready flag, a worker thread with an odd ID immediately passes the stop barrier to hit the subsequent start barrier. Each thread with an even ID n waits for thread $n+1$ to complete before it either passes the stop barrier itself if its ID is not a multiple of four, or it continues waiting for thread $n+2$ otherwise, and so on.

The basic realization of the stop barrier has been refined several times. For example, threads which synchronize with multiple other threads may do so in any order. In the case of `fold`-WITH-loops, final folding operations are integrated into the stop barrier and performed as soon as two threads synchronize with each other. This solution interleaves synchronization overhead with productive computations.

Implementing thread creation

In the enhanced fork/join model the performance impact of thread creation decreases with growing program execution time. Nevertheless, it makes sense in principle to care for efficient implementations. In a straightforward approach, the master thread creates all worker threads, one after the other. Execution of productive code is delayed by a time which grows linearly with the number of threads. We can reduce this initial delay by having the worker threads participate in thread creation. With a binary tree thread creation scheme, the initial delay is only $\mathcal{O}(log$ `NUM_THREADS`$)$.

Yet, we can do even better by exploiting the observation that many programs have a sequential startup phase. For example, they read input data from files or they

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

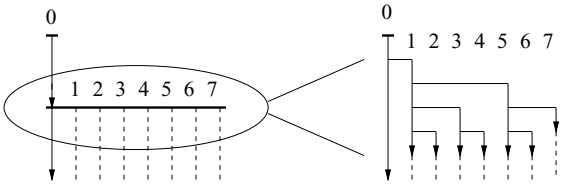

Fig. 14. Organization of thread creation phase.

initialize data structures. Therefore, it makes sense to exclude the master thread from worker thread creation. As sketched out in Figure 14, the master thread only creates a single worker thread and immediately starts execution of productive code. Instead of the master thread, the first worker thread initiates a binary tree thread creation scheme. Thread creation almost completely overlaps with a program's sequential startup phase and effectively reduces the thread creation overhead to $\mathcal{O}(1)$.

6 SPMD optimization

The enhanced fork/join execution model significantly reduces synchronization costs by replacing thread creation and thread termination by less expensive start and stop barriers, respectively. Nevertheless, program execution stalls at each stop barrier until arrival of the longest-running worker thread. Start and stop barriers are still major sources of overhead. In this section we describe optimization techniques which aim at eliminating synchronization barriers. Besides avoiding the cost immediately associated with the execution of the barrier code and the need to wait for the longest-running worker thread, larger regions of parallel execution also render the scheduling techniques discussed in section 4 more effective.

Introducing SPMD skeletons

So far, WITH-loops are used to describe both a computational task and a coordination behaviour, i.e. the organization of the parallel execution of the given task by multiple threads. Creating regions of parallel execution that stretch over several WITH-loops requires explicit separation of these two aspects. Therefore, we introduce *SPMD skeletons* as intermediate representations of the coordination behaviour of regions of parallel execution. Within such regions, which may contain multiple WITH-loops, program execution follows the "Single Program, Multiple Data" approach, hence the name. Our SPMD skeletons have the form

`spmd(` *USE, MAP, FOLD, CODE*`).`

The fourth parameter *CODE* refers to a sequence of WITH-loops embedded in a joint region of parallel execution without synchronization and communication inside. The other three parameters provide the necessary information to generate multi-threaded code from the `spmd` skeleton without identifying the WITH-loops inside. The first parameter *USE* contains the set of argument variables, which have to be communicated to worker threads when initiating parallel program execution. The second parameter *MAP* denotes a set of pairs each consisting of the result variable of an embedded `genarray`-WITH-loop and the associated shape specification. Similarly,

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

$$\mathcal{SPMD}\left[\!\!\left[\begin{array}{l}\texttt{a = with}\\ \quad gen_1 \texttt{ : } expr_1\\ \quad\quad\vdots\\ \quad gen_n \texttt{ : } expr_n\\ \texttt{genarray(} shp\texttt{);}\\ Rest\end{array}\right]\!\!\right]$$

$$\Longrightarrow \texttt{a = spmd(}\{shp\} \cup \bigcup_{j=1}^{n} FV(gen_j : expr_j),$$
$$\{\ [a, shp]\ \},$$
$$\{\ \},$$
$$\begin{array}{l}\texttt{a = with}\\ \quad gen_1 \texttt{ : } expr_1\\ \quad\quad\vdots\\ \quad gen_n \texttt{ : } expr_n\\ \texttt{genarray(} shp\texttt{););}\end{array}$$
$$\mathcal{SPMD}[\![Rest]\!]$$

$$\mathcal{SPMD}\left[\!\!\left[\begin{array}{l}\texttt{a = with}\\ \quad gen_1 \texttt{ : } expr_1\\ \quad\quad\vdots\\ \quad gen_n \texttt{ : } expr_n\\ \texttt{fold(} fold\_op\texttt{, } neutral\texttt{);}\\ Rest\end{array}\right]\!\!\right]$$

$$\Longrightarrow \texttt{a = spmd(}\{neutral\} \cup \bigcup_{j=1}^{n} FV(gen_j : expr_j),$$
$$\{\ \},$$
$$\{\ [a, fold\_op\ ]\ \},$$
$$\begin{array}{l}\texttt{a = with}\\ \quad gen_1 \texttt{ : } expr_1\\ \quad\quad\vdots\\ \quad gen_n \texttt{ : } expr_n\\ \texttt{fold(} fold\_op\texttt{, } neutral\texttt{););}\end{array}$$
$$\mathcal{SPMD}[\![Rest]\!]$$

$$\mathcal{SPMD}[\![\ a \texttt{ = } expr\texttt{;}\ \ Rest]\!] \quad\Longrightarrow\quad a \texttt{ = } expr\texttt{;}\ \ \mathcal{SPMD}[\![Rest]\!]$$

$$\mathcal{SPMD}[\![\texttt{return(}\ a_0\texttt{, }\ldots\texttt{, }a_k\texttt{);}]\!] \quad\Longrightarrow\quad \texttt{return(}\ a_0\texttt{, }\ldots\texttt{, }a_k\texttt{);}$$

Fig. 15. Embedding WITH-loops within `spmd` skeletons.

the third parameter *FOLD* denotes a set of pairs each consisting of the result variable of an embedded `fold`-WITH-loop and the associated fold operation. Figure 15 shows the transformation scheme $\mathcal{SPMD}$, which introduces `spmd` skeletons around individual WITH-loops.

Merging SPMD skeletons

Introducing `spmd` skeletons around WITH-loops alone does not alter the multi-threaded code generated. We merely lay the foundation for a subsequent optimization step, which tries to merge several `spmd` skeletons into a single one. This optimization is formalized by the transformation scheme $\mathcal{MERGE}$, which is shown in Figure 16. The definition of $\mathcal{MERGE}$ is based on guarded transformation rules. Predicate

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

$$\mathcal{MERGE}\left[\begin{array}{l} A = \texttt{spmd(}\ USE_A,\ MAP_A,\ FOLD_A,\ CODE_A\texttt{);} \\ B = \texttt{spmd(}\ USE_B,\ MAP_B,\ FOLD_B,\ CODE_B\texttt{);} \\ Rest \end{array}\right]$$

$$\Longrightarrow \mathcal{MERGE}\left[\begin{array}{l} A,\ B = \texttt{spmd(}\ USE_A \cup\ USE_B, \\ \qquad\qquad\quad MAP_A \cup\ MAP_B, \\ \qquad\qquad\quad FOLD_A \cup\ FOLD_B, \\ \qquad\qquad\quad CODE_A;\ CODE_B\texttt{);} \\ \qquad\qquad\quad Rest \end{array}\right]$$

$$\mid\ A \cap USE_B = \emptyset = A \cap B$$

$$\Longrightarrow A = \texttt{spmd(}\ USE_A,\ MAP_A,\ FOLD_A,\ CODE_A\texttt{);}\ \mathcal{MERGE}\left[\begin{array}{l} B = \texttt{spmd(}\ USE_B,\ MAP_B,\ FOLD_B,\ CODE_B\texttt{);} \\ Rest \end{array}\right]$$

$$\mid\ OTHERWISE$$

Fig. 16. SPMD optimization scheme $\mathcal{MERGE}$.

```
A = with ... D ... k ... C ... genarray( shp_A);
B = with ... C ... x ... y ... genarray( shp_B);
d = with ... k ... x ... z ... fold( fun, neutr);

                ⇓                ⇓                ⇓

A = spmd( {shp_A, D, k, C},
          {[A, shp_A]},
          { },
          A = with ... D ... k ... C ... genarray( shp_A); );
B = spmd( {shp_B, C, x, y},
          {[B, shp_B]},
          { },
          B = with ... C ... x ... y ... genarray( shp_B); );
d = spmd( {neutr, k, x, z},
          { },
          {[d, fun]},
          d = with ... k ... x ... z ... fold( fun, neutr); );

                ⇓                ⇓                ⇓

A, B, d = spmd( {shp_A, shp_B, D, k, C, x, y, z, neutr},
                {[A, shp_A], [B, shp_B]},
                {[d, fun] },
                A = with ... D ... k ... C ... genarray( shp_A);
                B = with ... C ... x ... y ... genarray( shp_B);
                d = with ... k ... x ... z ... fold( fun, neutr); );
```

Fig. 17. Example illustrating the SPMD optimization.

expressions following vertical bars restrict transformation rules to specific conditions. The predicate "otherwise" matches whenever none of the previous rules was applicable. $\mathcal{MERGE}$ identifies pairs of `spmd` skeletons which are directly adjacent in a sequence of assignments and which are free of data dependencies. Such pairs are combined into a single `spmd` skeleton with multiple return values. An example which illustrates both the introduction of `spmd` skeletons and the merging step can be found in Figure 17. For reasons of simplicity, only the free variables of the generator-expression pairs are shown rather than complete WITH-loops.

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

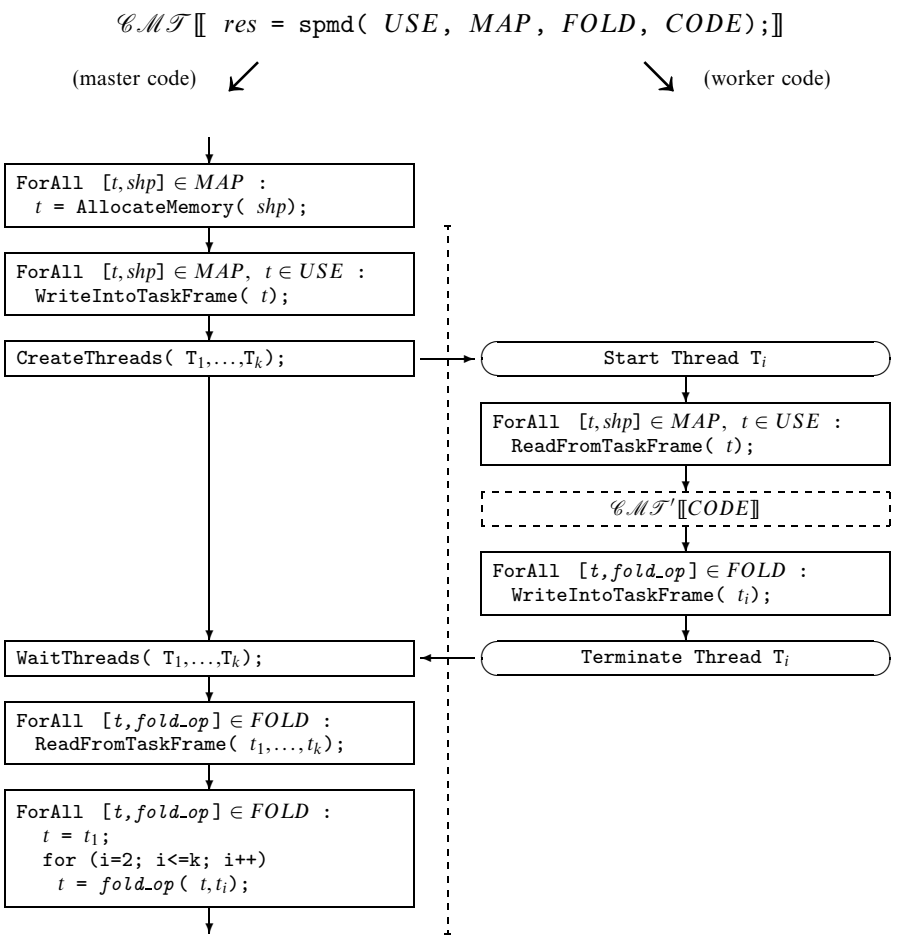

Fig. 18. Compilation scheme for `spmd` skeleton.

Compilation of SPMD skeletons

The SPMD optimization may lead to complex `spmd` skeletons containing many different WITH-loops. Nevertheless, the compilation schemes described in section 3 can be adapted to `spmd` skeletons almost straightforwardly. As shown in Figure 18, compilation of `spmd` skeletons mostly combines elements from Figure 3 and from Figure 6. In order to facilitate comparison of the new scheme with those presented in section 3, the terminology is based on the simple fork/join model rather than on the enhanced version.

Evaluation of an `spmd` skeleton starts with the allocation of memory for the result arrays of its embedded `genarray`-WITH-loops. Both their identifiers as well as their shapes are derived from the `MAP` set. Afterwards, all variables of the `USE` and `MAP` sets are written to the task frame. After creation, each worker thread immediately sets up its local execution environment for all WITH-loops within the `spmd` skeleton.

WITH-loops from the *CODE* section of the `spmd` skeleton are compiled using the slightly modified scheme $\mathscr{CMT}'$, which is shown in Figure 19. For each individual

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

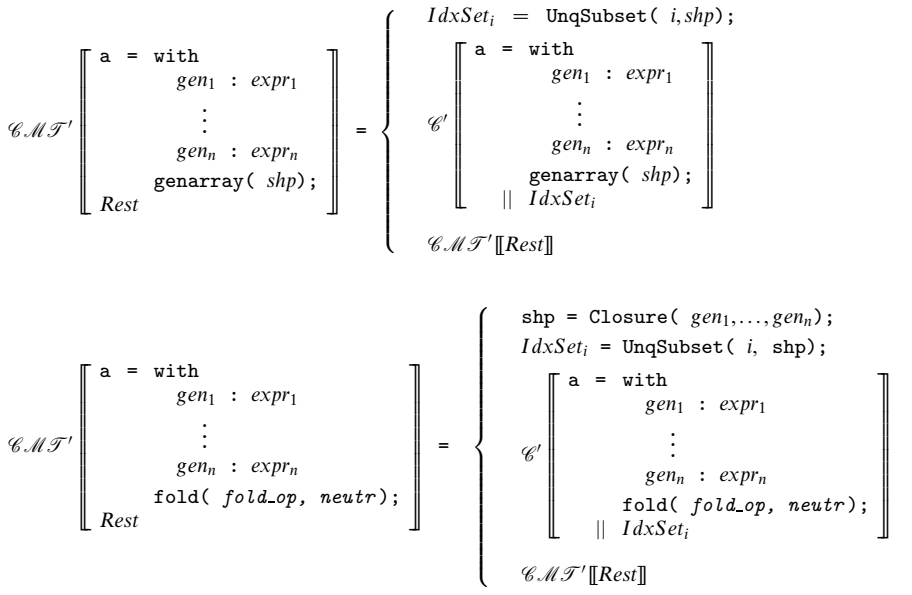

Fig. 19. Worker thread compilation scheme for `spmd` skeleton.

WITH-loop, code is generated which first identifies a unique index subspace and then performs the given numerical operation in a way that restricts all computations to exactly this subspace. Before termination, each worker thread sends its partial fold results, i.e. the variables of the `FOLD` set, back to the master thread. Eventually, the master thread folds the partial results to obtain the final values.

Enhancing the SPMD optimization by code restructuring

The optimization scheme $\mathcal{MERGE}$ discussed so far is limited to immediately adjacent `spmd` skeletons. Unfortunately, typical intermediate SAC codes interleave WITH-loops with scalar computations; adjacent `spmd` skeletons are rare. In order to improve the effectiveness of the transformation we extend $\mathcal{MERGE}$ by a code restructuring component. Our enhanced version of $\mathcal{MERGE}$ aims at moving scalar code between two consecutive `spmd` skeletons either ahead of the first skeleton or behind the second skeleton. However, both data dependencies and anti-dependencies restrict opportunities for code movement. Data dependencies reflect the nature of the problem and, hence, cannot be eliminated. In contrast, anti-dependencies arise from incidentally giving two conceptually different variables the same name. Anti-dependencies could be avoided by consistent variable renaming, but such a general code transformation is beyond the scope of this paper. Therefore, we take anti-dependencies into account in the sequel. Both data dependencies and anti-dependencies can be direct or indirect, as illustrated in the following example:

```
a = spmd( {u,c}, ... ) ;
b = ... a ... ;
c = ... d ... ;
d = spmd( {b,d}, ...) ;
```

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

Ignoring the scalar code between the two `spmd` skeletons makes them perfect candidates for merging. However, merging is prohibited by an indirect data dependency between the two skeletons via the scalar variable `b`. The assignment defining `c` also may not be moved, despite the absence of data dependencies. Both moving it ahead of the first or behind the second skeleton would penetrate the binding structure of the code fragment.

Figure 20 shows a refined version of the optimization scheme $\mathcal{MERGE}$, which – based on a thorough analysis of both direct and indirect dependencies and anti-dependencies – reorganizes the code as necessary to merge `spmd` skeletons whenever possible. Despite the restrictions discussed above, code reorganization is often feasible. To do so in a single sweep, three auxiliary parameters *store*, *use*, and *def* temporarily store scalar assignments and keep track of associated data dependencies and anti-dependencies; application of $\mathcal{MERGE}$ starts with all three being empty.

Leading scalar code is traversed by $\mathcal{MERGE}$ without alteration (3rd rule). The interesting case is encountered when the first `spmd` skeleton is reached during code traversal. Let us assume, the first skeleton is followed by a scalar assignment (2nd rule). If neither data dependencies nor anti-dependencies exist between them, the two assignments are exchanged. We keep the chance to merge the `spmd` skeleton with some subsequent one. Otherwise, it may still be possible to push the scalar assignment further down behind a subsequent skeletal assignment, but whether this will be possible with respect to data dependencies or whether another `spmd` skeleton follows at all is currently unknown. Therefore, the decision is postponed by temporarily appending the scalar assignment to the auxiliary store. To keep track of all data dependencies involving assignments currently residing in the auxiliary store, two variable sets *use* and *def* are maintained. Assuming another scalar assignment follows, it becomes clear that the decision whether this can be moved ahead of the preceding skeletal one involves all assignments in the auxiliary store.

Two skeletal assignments which directly follow each other or which have been made so by preceding transformations may or may not be merged. Once again, this depends on the auxiliary store (1st rule). In the absence of dependencies and anti-dependencies the two `spmd` skeletons are merged exactly as by the initial version of $\mathcal{MERGE}$. However, if merging is not possible, all scalar assignments from the auxiliary store are re-introduced into the sequence of assignments in between the two skeletons and the optimization scheme continues with the second skeletal assignment. The auxiliary store is flushed when $\mathcal{MERGE}$ reaches a `return`-statement (4th rule).

Discussion

As pointed out before, the SPMD optimization solely addresses WITH-loops in a sequence of assignments. Nestings of WITH-loops are eliminated beforehand as far as possible by WITH-LOOP-SCALARIZATION (Grelck *et al.*, 2004), a SAC-specific optimization technique. For remaining WITH-loop nestings, the recursive unfolding of parallel activity is prevented, as described in Section 3. Problems caused by the nesting of algorithmic skeletons are avoided.

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

(1) $\mathcal{MERGE}[\![\ A_1,\ldots,A_n$ = spmd(USE_A, MAP_A, $FOLD_A$, $CODE_A$);
$B_1,\ldots,B_m$ = spmd(USE_B, MAP_B, $FOLD_B$, $CODE_B$);
$Rest$ $]\!]$
$[\![store]\!][\![use]\!][\![def]\!]$

$\Longrightarrow \mathcal{MERGE}[\![\ A_1,\ldots,A_n,B_1,\ldots,B_m$ = spmd($USE_A \cup USE_B$,
$MAP_A \cup MAP_B$,
$FOLD_A \cup FOLD_B$,
$CODE_A$; $CODE_B$);
$Rest$ $]\!]$
$[\![store]\!][\![use]\!][\![def]\!]$

| $\emptyset = (\{A_1,\ldots,A_n\} \cup def) \cap USE_B$
| $\emptyset = (\{A_1,\ldots,A_n\} \cup def \cup use) \cap \{B_1,\ldots,B_m\}$

$\Longrightarrow A_1,\ldots,A_n$ = spmd(USE_A, MAP_A, $FOLD_A$, $CODE_A$);
$store$
$\mathcal{MERGE}[\![\ B_1,\ldots,B_m$ = spmd(USE_B, MAP_B, $FOLD_B$, $CODE_B$);
$Rest$ $]\!]$
$[\![\ ]\!][\![\ ]\!][\![\ ]\!]$

| $OTHERWISE$

(2) $\mathcal{MERGE}[\![\ A_1,\ldots,A_n$ = spmd(USE_A, MAP_A, $FOLD_A$, $CODE_A$);
$B_1,\ldots,B_m$ = $expr$;
$Rest$ $]\!]$
$[\![store]\!][\![use]\!][\![def]\!]$

$\Longrightarrow B_1,\ldots,B_m$ = $expr$;
$\mathcal{MERGE}[\![\ A_1,\ldots,A_n$ = spmd(USE_A, MAP_A, $FOLD_A$, $CODE_A$);
$Rest$ $]\!]$
$[\![store]\!][\![use]\!][\![def]\!]$

| $\emptyset = (\{A_1,\ldots,A_n\} \cup def) \cap USE_{expr}$
| $\emptyset = (\{A_1,\ldots,A_n\} \cup USE_{expr} \cup use) \cap \{B_1,\ldots,B_m\}$

$\Longrightarrow \mathcal{MERGE}[\![\ A_1,\ldots,A_n$ = spmd(USE_A, MAP_A, $FOLD_A$, $CODE_A$);
$Rest$ $]\!]$
$[\![store\ B_1,\ldots,B_m$ = $expr$; $]\!][\![use \cup USE_{expr}]\!][\![def \cup B]\!]$

| $OTHERWISE$

(3) $\mathcal{MERGE}[\![\ A_1,\ldots,A_n$ = $expr$; $Rest$ $]\!]$ $[\![store]\!][\![use]\!][\![def]\!]$

$\Longrightarrow A_1,\ldots,A_n$ = $expr$; $\mathcal{MERGE}[\![Rest]\!][\![$ `store` $]\!][\![$ `use` $]\!][\![$ `def` $]\!]$

(4) $\mathcal{MERGE}[\![$ return($A_1,\ldots,A_n$); $]\!][\![store]\!][\![use]\!][\![def]\!]$

$\Longrightarrow store$ return($A_1,\ldots,A_n$);

Fig. 20. Refined $\mathcal{MERGE}$ scheme with code restructuring capability.

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

```
for (i=0; i<max_iter; i+=1) {
  A = with
        ([  0,  0] <= iv < [  1,  N]) : B[iv]
        ([  1,  0] <= iv < [M-1,  1]) : B[iv]
        ([  1,  1] <= iv < [M-1,N-1]) : 0.25 * (B[iv+[1,0]] + B[iv-[1,0]]
                                                + B[iv+[0,1]] + B[iv-[0,1]])
        ([  1,N-1] <= iv < [M-1,  N]) : B[iv]
        ([M-1,  0] <= iv < [  M,  N]) : B[iv]
      genarray( [M,N]);
  B = A;
}
```

Fig. 21. Code fragment of benchmark `jacobi`.

The SPMD optimization bears some similarity to the fusion of adjacent collective operations in MPI (Gorlatch *et al.*, 1999) or to the elimination of synchronization barriers in the parallelization of imperative programs (O'Boyle *et al.*, 1995; Tseng, 1995; Han *et al.*, 1998). However, the often opaque data flow in imperative environments prevents large-scale code restructuring as we do.

7 Experimental evaluation

This section reports on a series of experiments evaluating the compilation techniques presented so far. In order to quantify the impact of individual design decisions on the parallel performance of compiled SAC code we restrict ourselves to a set of representative micro benchmarks. Readers looking for more general case studies on programming methodology and runtime performance in comparison with other programming environments are referred elsewhere (Grelck & Scholz, 2000; Grelck, 2002; Grelck & Scholz, 2003b; Grelck & Scholz, 2003a). Experiments have been made on three different machine architectures: a 4-processor SUN E650, a 12-processor SUN E4000, and a 72-processor SUN E15k. All machines run different versions of the SOLARIS operating system. Since access to the two larger machines has been non-exclusive, not all processors could effectively be used for the experiments.

Comparison of execution models

As a starting point and performance base line we investigate the parallel performance of a well-known benchmark kernel: 2-dimensional Jacobi relaxation with a 4-point stencil. Various ways of implementing Jacobi relaxation in SAC have been discussed in Grelck (2001) and Scholz (2003). Figure 21 shows the relevant fragment of intermediate SAC code compiled from various different high-level specifications.

Basically, a new array `A` is computed from an existing array `B` by setting all inner elements to the arithmetic mean of their four direct neighbours while copying the boundary elements. A single multi-generator WITH-loop suffices as internal representation. Embedding the WITH-loop within a sequential FOR-loop reflects the iterative nature of relaxation. For reasons of simplicity, we use a fixed number of iterations rather than a convergence criterion. This intermediate SAC code can be

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

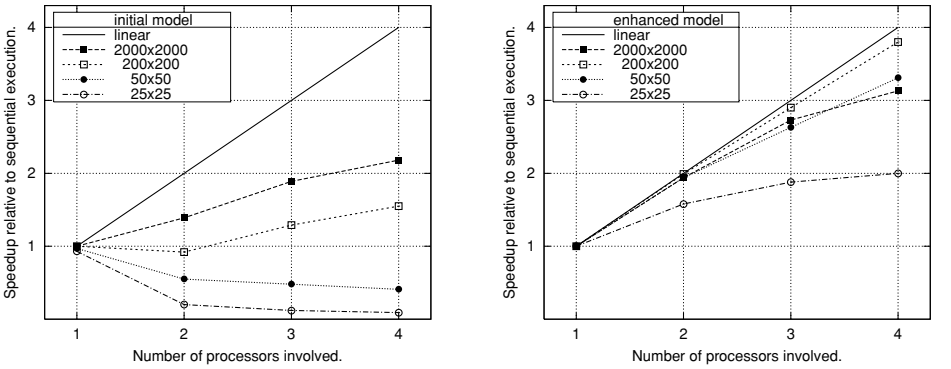

Fig. 22. Speedups observed for different problem sizes of benchmark `jacobi` on SUN E650 with parallelization according to the simple fork/join model (left) and to the enhanced fork/join model (right).

compiled into C code which is competitive with hand-optimized implementations in low-level imperative languages (Grelck, 2001; Scholz, 2003).

Despite its simplicity, `jacobi` is not a trivial benchmark as it does require synchronization and data exchange between processors after each iteration. In this important aspect it differs from other popular benchmarks like ray tracing, Mandelbrot sets, or matrix multiplication. Jacobi relaxation is particularly suitable for our purpose because we can manipulate the ratio between computation and synchronization/communication by simple modification of the array size.

Figure 22 compares speedups achieved by parallelization according to the initial, pure fork/join execution model, described in section 3, with those obtained by using the enhanced fork/join execution model, outlined in section 5. All speedup figures are given relative to sequential execution time of the same SaC code. To compare both execution models on a fair basis, we allow the master thread to participate in parallel execution of WITH-loops even in the initial model.

As expected, the organizational overhead inflicted by thread creation and termination in the initial execution model is prohibitively high. For small problem sizes and, hence, high synchronization and communication demands, severe slowdowns must be observed. Even with a large array size of 2000 by 2000 elements and, hence, relatively infrequent synchronization and communication events, the simple fork/join model dissatisfies with a maximum speedup of only 2.2 with four processors. In contrast, the enhanced fork/join model achieves almost linear speedups. Even for array sizes as small as 25 by 25 elements and, hence, extremely frequent synchronization and communication events four processors achieve a speedup of 2.0, not a slowdown.

This experiment also demonstrates the sensitivity of sequentially competitive code to certain design decisions in parallelization. With arrays of 200 by 200 elements and two processors parallelization according to the simple fork/join model yields almost the same execution time as sequential operation. This means that the overhead of creating and terminating a single worker thread in each iteration equals the entire runtime of the productive code. Consider as an example the productive code would be ten times slower. Thread creation and termination take the same absolute

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

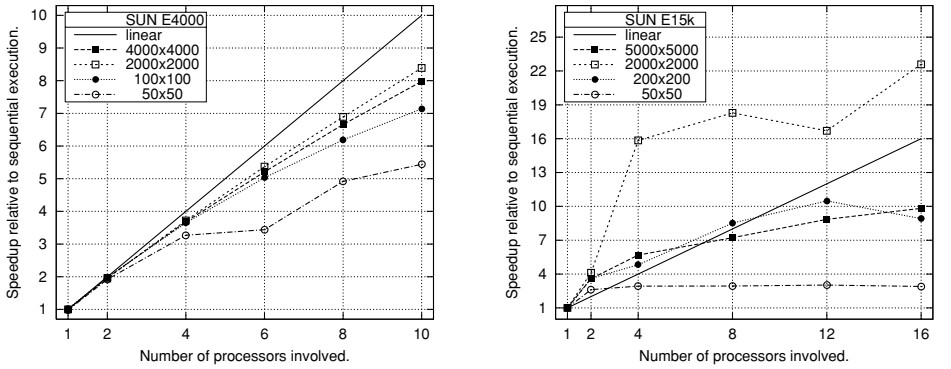

Fig. 23. Speedups observed for different problem sizes of benchmark `jacobi` on SUN E4000 (left) and SUN E15k (right) with parallelization according to the enhanced fork/join model.

time as before. Suddenly, an excellent speedup of 1.9 could be observed for the simple fork/join model. The enhanced fork/join model would still yield a speedup of 2.0, just as before. Without any modification to the parallelization techniques, the – obviously false – impression would be that in practice both execution models yield similar efficiency. This example illustrates both the importance of sequentially competitive code for analysis and evaluation of parallelization techniques and the particular challenge to actually achieve reasonable speedups by parallelization of competitive sequential code.

Since the simple fork/join model does not even scale on the smallest of our three test systems, all further experiments use the enhanced execution model. Figure 23 shows speedups for `jacobi` on the SUN E4000 and on the SUN E15k. Scalability carries over to larger processor counts. With infrequent synchronization and communication, parallel performance is mostly limited by the memory bandwidth constraints of the different architectures. The fact that reasonable speedups are obtained even for arrays as small as 50 by 50 elements demonstrates the efficiency of our implementation in the context of extremely frequent synchronization and communication.

Superlinear speedups observed for some problem sizes must be attributed to cache effects. Employing additional processors increases the amount of cache memory. For certain combinations of problem size and cache configuration, taking one additional processor lets all memory accessed by one processor fit into its local caches. The abruptly improved data locality results in a performance boost.

Evaluation of the SPMD optimization

Since Jacobi relaxation requires synchronization after each iteration step, it is not suitable to analyze the performance impact of our SPMD optimization. Instead, we use two micro benchmarks called `sequence` and `triangular`.

As shown in Figure 24, the micro benchmark `sequence` consists of a loop containing a sequence of five `genarray`-WITH-loops. Each WITH-loop creates a vector of `N` integers initialized to a constant value. In the absence of data dependencies,

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

```
for (i=0; i<M; i+=1) {
  A = with
        ([0] <= iv < [N]) : 1
      genarray( [N]);
  B = with
        ([0] <= iv < [N]) : 2
      genarray( [N]);
  C = with
        ([0] <= iv < [N]) : 3
      genarray( [N]);
  D = with
        ([0] <= iv < [N]) : 4
      genarray( [N]);
  E = with
        ([0] <= iv < [N]) : 5
      genarray( [N]);
}
```

Fig. 24. Code fragment of benchmark `sequence`.

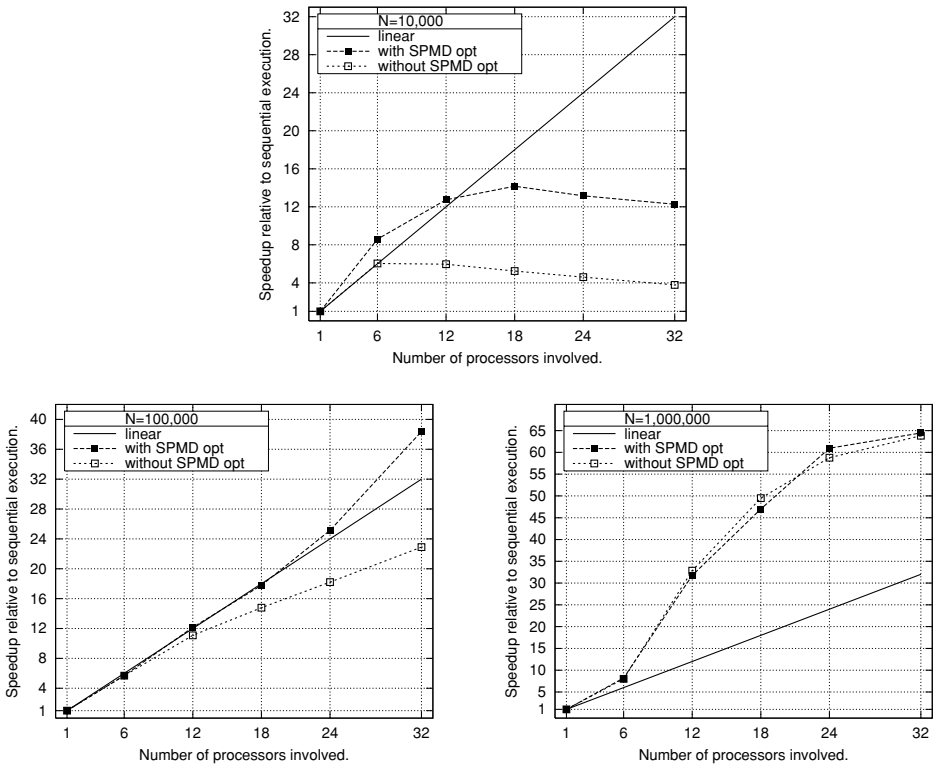

Fig. 25. Effect of the SPMD optimization on speedups achieved by benchmark `sequence` for different problem sizes on SUN E15k.

the SPMD optimization manages to merge the initially five SPMD skeletons into a single one and, hence, to remove four out of five synchronization barriers in compiled code. The effect on parallel performance is shown in Figure 25 for different vector sizes `N`. As expected, the impact of the SPMD optimization on runtime performance

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

```
A = with
      ([0] <= iv < [M]) : with
                            ([0] <= jv <  iv) : 0
                            ( iv <= jv < [N]) : expensive(iv,jv)
                          genarray( [N])
    genarray( [M]);

B = with
      ([0] <= iv < [M]) : with
                            ([0] <= jv <  iv) : expensive(iv,jv)
                            ( iv <= jv < [N]) : 0
                          genarray( [N])
    genarray( [M]);
```

Fig. 26. Code fragment of benchmark `triangular`.

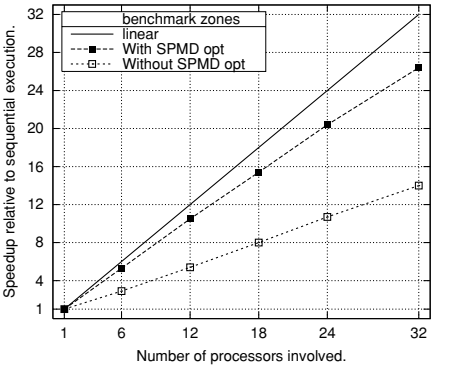

Fig. 27. Effect of SPMD optimization on speedups achieved by benchmark `triangular` on SUN E15k.

grows with program execution being increasingly dominated by synchronization overhead, which in turn grows with increasing processor counts.

The micro benchmark `triangular`, as shown in Figure 26, maps a complex numerical operation to an upper triangular matrix and, in a subsequent step, to a lower triangular matrix. The purpose of `triangular` is to demonstrate the potential impact of the SPMD optimization on workload distribution.

Using the default WITH-loop scheduling, workload is poorly balanced in both individual steps. However, if we manage to eliminate the intermediate synchronization barrier, workload imbalances should vanish. Figure 27 shows the observable impact of the SPMD optimization on speedups achieved by `triangular` for a matrix size of 1000 by 1000 elements each. Speedups almost double with SPMD optimization switched on, although the frequency of synchronization is insignificant due to the large size of the matrices involved.

Although the SAC programs implementing the two micro benchmarks are rather simple, it is noteworthy that no optimization would have been possible at all by using only the simple version of the $\mathcal{MERGE}$ scheme. Preceding code transformations and optimizations performed by the SAC compiler always introduce scalar code in between WITH-loops. This demonstrates the need for the code restructuring version of $\mathcal{MERGE}$, as described in section 6.

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

```
M = N / 8;
A = with
      ([  0] <= iv < [  M]) : fun(1)
      ([  M] <= iv < [2*M]) : fun(2)
      ([2*M] <= iv < [3*M]) : fun(4)
      ([3*M] <= iv < [4*M]) : fun(8)
      ([4*M] <= iv < [5*M]) : fun(16)
      ([5*M] <= iv < [6*M]) : fun(32)
      ([6*M] <= iv < [7*M]) : fun(64)
      ([7*M] <= iv < [  N]) : fun(128)
    genarray( [N]);
```

Fig. 28. Code fragment of benchmark `zones`.

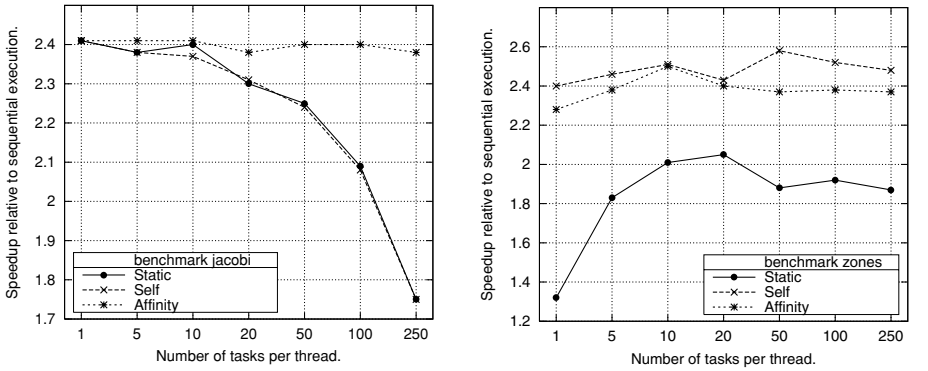

Fig. 29. Performance impact of different task sizes and task schedulers on benchmarks `jacobi` (left) and `zones` (right) using 3 threads on SUN E650.

Evaluation of with*-loop scheduling*

All experiments described so far have used the default WITH-loop scheduler `Static/Even(1)`, which realizes a static horizontal block decomposition. However, the opportunity to plug in different task selectors and different task schedulers, as explained in Section 4, offers a wide range for further experimental investigation. We use two different benchmarks for this. Jacobi relaxation, as introduced in the beginning of this section, serves as a representative for programs with an even distribution of computational workload. In the following, we use a fixed problem size of 2000 by 2000 elements and 100 iterations. As a representative for irregular codes, the micro benchmark `zones` processes a vector which is divided into eight equally sized sections; the computational complexity per element doubles from one section to the next. The relevant code fragment of `zones` is shown in Figure 28.

Figure 29 illustrates the performance impact of different task sizes for all three task schedulers introduced in section 4. For the regular code of benchmark `jacobi` a small number of tasks turns out to be more advantageous than a large number. As active workload balancing is not necessary, large numbers of tasks only contribute to organizational overhead. Both `Static` and `Self` task scheduling additionally suffer from decreasing data locality. In contrast, `Affinity` task scheduling retains almost constant performance levels. For a regular distribution of workload, as in the case

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

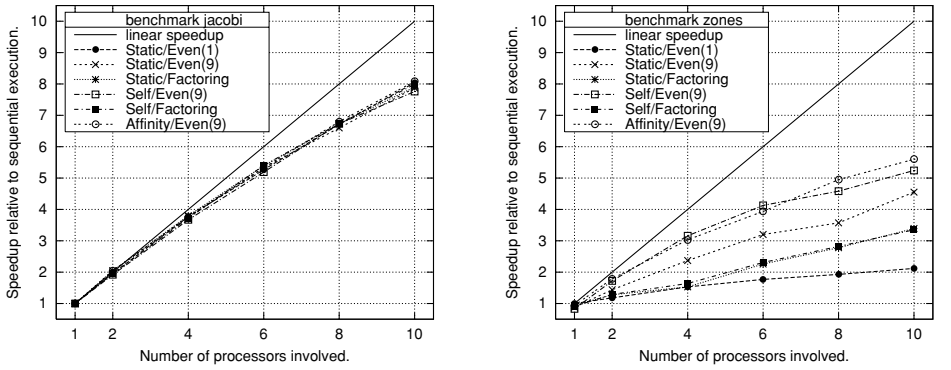

Fig. 30. Impact of different combinations of task selector and task scheduler on the scalability of benchmarks `jacobi` (left) and `zones` (right) on SUN E4000.

of `jacobi`, affinity scheduling never reaches the work stealing phase. As long as the static a-priori assignment of work to threads suffices, `Affinity` produces very low overhead.

In the less regular scenario of benchmark `zones`, larger numbers of tasks generally improve the dynamic load balancing capabilities of the task schedulers `Self` and `Affinity`. However, over a certain threshold, the positive impact of improved load balancing is outweighed by increasing overhead. These contrary effects result in a rather small, but still observable, performance impact of the task size. Since the example is not sensitive to data locality, the task scheduler `Affinity` cannot exploit its specific capabilities and creates slightly more overhead than `Self`. The `Static` task scheduler achieves poor speedups with a single task per thread. With increasing number of tasks per thread, `Static` results in a block-cyclic and, eventually, in a cyclic scheduling, which better suits the distribution of computational workload in the benchmark `zones`. Again, after reaching a certain level of workload balancing, additional increase in the number of tasks creates more overhead than is outweighed by an even smoother workload distribution.

Figure 30 shows the parallel performance of the benchmarks `jacobi` and `zones` with different combinations of task selectors and task schedulers. In this experiment, we use nine tasks per thread as a compromise to remove one degree of freedom. It turns out that the performance differences in the regular case are extremely small. Table 2 provides exact figures for ten processors. Minor differences to the data shown in Figure 23 are due to using different SAC implementations of Jacobi relaxation at different times and with slightly different machine configurations. For benchmark `zones`, the different load balancing capabilities of the task scheduler / task selector combinations become more apparent. While `Affinity` and `Self/Even(9)` as dynamic techniques yield the best results, `Static/Even(9)` achieves almost as good figures. This combination of task selector and task scheduler realizes a block-cyclic scheduling, which perfectly matches the remaining regularity in the example code. All other combinations turn out to be inappropriate in this scenario.

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

Table 2. *Speedups achieved by benchmarks* `jacobi` *and* `zones` *with different combinations of task selector and task scheduler using 10 processors on SUN E4000*

Task Scheduler	Task Selector	`jacobi`	`zones`
Static	Even(1)	8.04	2.12
Static	Even(9)	7.87	4.55
Static	Factoring	7.90	3.39
Self	Even(9)	7.76	5.24
Self	Factoring	7.99	3.35
Affinity	Even(9)	8.08	5.60

8 Conclusion

SaC is a purely functional array processing language designed with numerical applications in mind. SaC combines declarative programming on multi-dimensional stateless arrays with high runtime performance. Tailor-made optimization techniques restructure SaC code from a representation amenable to development and maintenance towards a representation suitable for efficient execution by machines. As a result of thorough optimization, the runtime behaviour of high-level SaC programs is often competitive with equivalent applications implemented in low-level imperative languages.

High sequential performance gives us the opportunity to achieve real performance gains over existing imperative implementations, when exploiting the conceptual advantages of the functional paradigm for parallelization. The particular challenge is that high sequential performance makes the realization of satisfying speedups much more difficult. Fast sequential code is bad for the ratio between organizational overhead and productive code. Even minor inefficiencies in the organization of parallel program execution substantially degrade scalability.

This paper describes compilation techniques and runtime system support for the implicit parallelization of SaC programs similar to a compiler optimization. We specifically target shared memory multiprocessors; our runtime system is based on Pthreads. Several case studies have shown that simple recompilation of SaC code yields speedups that are competitive with manually parallelized imperative programs. High sequential performance allows implicitly parallelized SaC code to substantially outperform even hand-optimized sequential imperative programs (Grelck & Scholz, 2000; Grelck, 2002; Grelck & Scholz, 2003b; Grelck & Scholz, 2003a).

SaC demonstrates the suitability of functional languages for numerically intensive application domains in principle. However, efficient support for processing multi-dimensional arrays must be a primary concern for language design, implementation, and parallelization. When these conditions are met, the functional paradigm has a lot to offer to numerical application programmers. Functional programs are generally more closely related to mathematical specifications of algorithms than their imperative counterparts. Narrowing the gap between algorithmic specification

https://doi.org/10.1017/S0956796805005538 Published online by Cambridge University Press

and executable code increases both programmer productivity and confidence in correctness. Even research labs and government agencies will not forever have the necessary manpower and relaxed time constraints to create larger and larger application programs based on low-level machine-oriented techniques. Confidence into program correctness is of specific interest to numerical applications because in many cases, e.g. simulations, hardly more than weak plausibility checks can be applied to examine the correctness of numerical results.

Another consequence of high-level programming is a loss of control over the intricacies of program execution. This prevents manual optimization and customization of code to concrete target machines. However, it is exactly this manual optimization and customization that takes a lot of time, effort, and expert knowledge when applying low-level, machine-centric programming techniques. Adaptation to specific machine requirements reduces readability and portability of code and seriously complicates code maintenance.

Abstract program specifications give compilers more freedom in code generation and often a better understanding of what is to be computed. The absence of side-effects and an explicit data flow enhance opportunities for large-scale code restructuring optimizations just as for truly compiler-directed parallelization. By utilization of moderate additional hardware resources, functional programs may significantly outperform imperative programs, as shown in this paper.

Although we have developed and presented our parallelization techniques in the context of the functional array language SaC, many ideas are not limited to SaC. They could be carried over to implementations of other functional languages which meet the essential requirement of the domain of numerical applications: efficient support for multi-dimensional numerical arrays.

Having shown that SaC programs can be compiled into efficiently executable parallel code for shared memory systems, our next step is to address distributed memory architectures. Finding out to which extent the experience gained with shared memory systems can be carried over to distributed memory systems will be both interesting and challenging. With dedicated implementations for both architectures, we would also provide a tailor-made solution for modern hybrid systems made up of clusters of large SMP nodes.

Acknowledgements

Many people have in one way or another contributed to the work described in this paper. First, I wish to thank my former PhD supervisor Werner Kluge for stimulating my interest into research and for giving me the exciting and challenging task that laid the foundations for this work. I'm grateful to my colleagues and friends Sven-Bodo Scholz and Dietmar Kreye for countless intense and fruitful discussions. I'm also indebted to my students Jan-Hendrik Schöler and Borg Enders, who contributed parts of the implementation and conducted some of the experiments. Last but not least, I wish to thank the anonymous reviewers for their detailed and valuable comments and Robert Bernecky for proof-reading the final manuscript.

References

Achten, P. and Plasmeijer, M. J. (1995) The ins and outs of Clean I/O. *J. Funct. Program.* **5**(1), 81–110.

Adams, J. C., Brainerd, W. S., Martin, J. T., Smith, B. T. and Wagener, J. L. (1997) *Fortran-95 Handbook – Complete ANSI/ISO Reference*. Scientific and Engineering Computation. MIT Press.

Allen, R. and Kennedy, K. (2001) *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann.

Bailey, D. and Newey, M. (1993) An extension of ML for distributed memory computers. *Proceedings 16th Australasian Computer Science Conference*, pp. 387–396. Brisbane, Australia.

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, T. A., Simon, R. S., Venkatakrishnam, V. and Weeratunga, S. K. (1991) The NAS Parallel Benchmarks. *Int. J. Supercomput. Applic.* **5**(3), 63–73.

Barendsen, E. and Smetsers, S. (1995) Uniqueness Type Inference. In: Hermenegildo, M. and Swierstra, S. D., editors, *Proceedings 7th International Symposium on Programming Language Implementation and Logic Programming (PLILP'95): Lecture Notes in Computer Science 982*, pp. 189–206. Utrecht, The Netherlands. Springer-Verlag.

Bernecky, R. (1997) *APEX: The APL Parallel Executor*. MPhil thesis, University of Toronto, Toronto, Canada.

Blelloch, G. E. and Sabot, G. W. (1990) Compiling Collection-Oriented Languages onto Massively Parallel Computers. *J. Parallel & Distributed Comput.* **8**(2), 119–134.

Blelloch, G. E., Chatterjee, S., Hardwick, J. C., Sipelstein, J. and Zagha, M. (1994) Implementation of a Portable Nested Data-Parallel Language. *J. Parallel & Distributed Comput.* **21**(1), 4–14.

Blelloch, G. E. (1996) Programming parallel algorithms. *Comm. ACM*, **39**(3).

Bratvold, T. A. (1993) A skeleton-based parallelising compiler for ML. In: Plasmeijer, M. J. and van Eekelen, M., editors, *Proceedings 5th International Workshop on Parallel Implementations of Functional Languages*, pp. 23–34. Nijmegen, The Netherlands.

Breitinger, S., Loogen, R., Ortega-Mallén, Y. and Peña, R. (1997) The Eden Coordination Model for distributed memory systems. *Proceedings Conference on High-level Parallel Programming Models and Supportive Environments (HIPS'97)*, Geneva, Switzerland. IEEE Press.

Bülck, T., Held, A., Kluge, W. E., Pantke, S., Rathsack, C., Scholz, S.-B. and Schröder, R. (1994) Experience with the implementation of a concurrent graph reduction system on an ncube/2 platform. In: Buchberger, B. and Volkert, J., editors, *Proceedings Joint International Conference on Parallel and Vector Processing: Lecture Notes in Computer Science 854*, pp. 497–508. Springer-Verlag.

Cann, D. C. (1989) *Compilation Techniques for High Performance Applicative Computation*. Technical report CS-89-108, Lawrence Livermore National Laboratory, Livermore, California, USA.

Cann, D. C. (1992) Retire Fortran? A debate rekindled. *Comm. ACM*, **35**(8), 81–89.

Cann, D. C. and Evripidou, P. (1995) Advanced Array Optimizations for High Performance Functional Languages. *IEEE Trans. Parallel & Distributed Syst.* **6**(3), 229–239.

Chakravarty, M. M. T. and Keller, G. (2003) An approach to fast arrays in Haskell. In: Jeuring, J. and Peyton Jones, S., editors, *Summer School and Workshop on Advanced Functional Programming: Lecture Notes in Computer Science 2638*, pp. 27–58. Springer-Verlag.

Chamberlain, B. L., Choi, S.-E., Lewis, C., Snyder, L., Weathersby, W. D. and Lin, C. (1998) The Case for High-Level Parallel Programming in ZPL. *IEEE Computational Sci. & Eng.* **5**(3).

Cohen, W. E., Dietz, H. G. and Sponaugle, J. B. (1994) *Dynamic Barrier Architecture for Multi-Mode Fine-Grain Parallelism using Conventional Processors*. Technical report, TR-EE 94-9, School of Electrical Engineering, Purdue University, Indiana, USA.

Cole, M. I. (1989) *Algorithmic Skeletons: Structured Management of Parallel Computation*. Reserach Monographs in Parallel and Distributed Computing. Pitman.

Cole, M. (2004) Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.* **30**(3), 389–406.

Cooper, E. C. and Morrisett, J. (1990) *Adding Threads to Standard ML*. Technical report, CMU-CS-90-186, Carnegie-Mellon-University, School of Computer Science, Pittsburgh, Pennsylvania, USA.

Dagum, L. and Menon, R. (1998) OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Trans. Computational Sci. & Eng.* **5**(1).

Darlington, J., Field, A. J., Harrison, P. G., Kelly, P. H. J., Sharp, D. W. N., Wu, Q. and While, R. L. (1993) Parallel Programming using Skeleton Functions. In: Bode, A., Reeve, M. and Wolf, G., editors, *Proceedings 5th Conference on Parallel Architectures and Languages Europe (PARLE'93): Lecture Notes in Computer Science 694*, pp. 146–160. Springer-Verlag.

Dongarra, J. J., Meuer, H. W. and Strohmaier, E. (2003) TOP500 Supercomputer Sites, 22nd edition. *Proceedings Supercomputing Conference (SC'03)*, Phoenix, AZ. ACM Press.

Feo, J. T., Miller, P. J., Skedzielewski, S. K., Denton, S. M. and Solomon, C. J. (1995) Sisal 90. In: Böhm, A. P. W. and Feo, J. T.editors, *Proceedings Conference on High Performance Functional Computing (HPFC'95)*, pp. 35–47. Denver, CO. Lawrence Livermore National Laboratory, Livermore, California, USA.

Gorlatch, S., Wedler, C. and Lengauer, C. (1999) Optimization rules for programming with collective operations. In: Atallah, M., editor, *Proceedings 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP'99)*, pp. 492–499. San Juan, Puerto Rico.

Grelck, C. (1999) Shared memory multiprocessor support for SAC. In: Hammond, K., Davie, T. and Clack, C., editors, *Proceedings 10th International Workshop on Implementation of Functional Languages (IFL'98): Lecture Notes in Computer Science 1595*, pp. 38–54. Springer-Verlag.

Grelck, C. (2001) *Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC — Single Assignment C*. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Kiel, Germany. Logos Verlag.

Grelck, C. (2002) Implementing the NAS Benchmark MG in SAC. In: Prasanna, V. K. and Westrom, G., editors, *Proceedings 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*, pp. 38–54. Fort Lauderdale, FL. IEEE Press.

Grelck, C. (2003) A multithreaded vompiler backend for high-level array programming. In: Hamza, M. H., editor, *Proceedings 21st International Multi-conference on Applied Informatics (AI'03), Part ii: International conference on parallel and distributed computing and networks (pdcn'03)*, pp. 478–484. Innsbruck, Austria. ACTA Press.

Grelck, C. and Scholz, S.-B. (1995) Classes and objects as basis for I/O in SAC. In: Johnsson, T., editor, *Proceedings 7th International Workshop on Implementation of Functional Languages (IFL'95)*, pp. 30–44. Båstad, Sweden. Gothenburg, Sweden.

Grelck, C. and Scholz, S.-B. (1999) Accelerating APL programs with SAC. In: Lefèvre, O., editor, *Proceedings International Conference on Array Processing Languages (APL'99)*, pp. 50–57. Scranton, PN. ACM Press.

Grelck, C. and Scholz, S.-B. (2000) HPF vs. SAC – A case study. In: Bode, A., Ludwig, T., Karl, W. and Wismüller, R., editors, *Proceedings 6th European Conference on Parallel Processing (EURO-PAR'00): Lecture Notes in Computer Science 1900*, pp. 620–624. Munich, Germany. Springer-Verlag.

Grelck, C. and Scholz, S.-B. (2003a) SAC – From high-level programming with arrays to efficient parallel execution. *Parallel Process. Lett.* **13**(3), 401–412.

Grelck, C. and Scholz, S.-B. (2003b) Towards an Efficient Functional Implementation of the NAS Benchmark FT. In: Malyshkin, V., editor, *Proceedings 7th International Conference on Parallel Computing Technologies (PACT'03): Lecture Notes in Computer Science 2763*, ppp. 230–235. Nizhni Novgorod, Russia. Springer-Verlag.

Grelck, C., Kreye, D. and Scholz, S.-B. (2000) On code generation for multi-generator WITH-Loops in SAC. In: Koopman, P. and Clack, C., editors, *Proceedings 11th International Workshop on Implementation of Functional Languages (IFL'99): Lecture Notes in Computer Science 1868*, pp. 77–94. Lochem, The Netherlands. Springer-Verlag.

Grelck, C., Scholz, S.-B. and Trojahner, K. (2004) With-loop scalarization: merging nested array operations. In: Trinder, P. and Michaelson, G., editors, *Proceedings 15th International Workshop on Implementation of Functional Languages (IFL'03): Lecture Notes in Computer Science 3145*. Springer-Verlag.

Haines, M. and Böhm, W. (1993) Task management, virtual shared memory, and multithreading in a distributed memory implementation of SISAL. In: Bode, A., Reeve, M. and Wolf, G., editors, *Proceedings 5th Conference on Parallel Architectures and Languages Europe (PARLE'93): Lecture Notes in Computer Science 694*, pp. 12–23. Springer-Verlag.

Hamdan, M., Michaelson, G. and King, P. (1999) A framework for nesting algorithmic skeletons. *Proceedings International Conference on Parallel Computing (PARCO'99)*, Delft, The Netherlands.

Hammes, J., Sur, S. and Böhm, W. (1997) On the effectiveness of functional language features: NAS Benchmark FT. *J. Funct. Program.* **7**(1), 103–123.

Hammond, K. (1994) Parallel functional programming: an introduction (invited paper). In: Hong, H., editor, *First International Symposium on Parallel Symbolic Computation (PASCO'94)*, pp. 181–193. Linz, Austria. World Scientific.

Hammond, K. and Portillo, A. J. R. (2000) HaskSkel: Algorithmic skeletons in Haskell. In: Clack, C. and Koopman, P., editors, *Proceedings 11th International Workshop on Implementation of Functional Languages (IFL'00): Lecture Notes in Computer Science 1868*, pp. 181–198. Lochem, The Netherlands. Springer-Verlag.

Han, H., Tseng, C.-W. and Keleher, P. (1998) Eliminating barrier synchronization for compiler-parallelized codes on software DSMs. *Int. J. Parallel Program.* **26**(5), 591–612.

Hartel, P. H. and Langendoen, K. G. (1993) Benchmarking implementations of lazy functional languages. *Proceedings Conference on Functional Programming Languages and Computer Architecture (FPCA'93)*, PP. 341–349. Copenhagen, Denmark. ACM Press.

Hartel, P. H. *et al.* (1996) Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *J. Funct. Program.* **6**(4).

Hennessy, J. L. and Patterson, D. A. (2003) *Computer Architecture: A quantitative approach, third edition*. Morgan Kaufmann.

Hill, J. M. D. and Skillicorn, D. B. (1998) Practical Barrier Synchronisation. *Proceedings 6th Euromicro Workshop on Parallel and Distributed Processing (PDOP'98)*, pp. 438–444. Barcelona, Spain. IEEE Press.

Hudak, P. and Bloss, A. (1985) The aggregate update problem in functional programming systems. *Proceedings 12th ACM Symposium on Principles of Programming Languages (POPL'85)*, pp. 300–313. New Orleans, LA. ACM Press.

Hummel, S. F., Schonberg, E. and Flynn, L. E. (1992) Factoring: A method for scheduling parallel loops. *Comm. ACM*, **35**(8), 90–101.

Institute of Electrical and Electronic Engineers, Inc. (1995) *Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]*. IEEE Standard 1003.1c–1995. IEEE, New York City, New York, USA. also ISO/IEC 9945-1:1990b.

Jones, S. L. Peyton, Gordon, A. and Finne, S. (1996) Concurrent Haskell. *Proceedings 23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pp. 295–308. St. Petersburg Beach, FL. ACM Press.

Kelly, P. and Taylor, F. (1999) Coordination Languages. In: Hammond, K. and Michaelson, G., editors, *Research Directions in Parallel Functional Programming*, pp. 305–321. Springer-Verlag.

Koebel, C. H., Loveman, D. B., Schreiber, R. S., Steele, G. L. and Zosel, M. E. (1994) *The High Performance Fortran Handbook*. MIT Press.

Kruskal, C. and Weiss, A. (1985) Allocating independent subtasks on parallel processors. *IEEE Trans. Softw. Eng.*, **11**(10), 1001–1016

Li, H., Tandri, S., Stumm, M. and Sevcik, K. C. (1993) Locality and loop scheduling on NUMA multiprocessors. *Proceedings 22nd International Conference on Parallel Processing (ICPP'93)*, pp. 140–147.Boca Raton, FL. CRC Press.

Loidl, H.-W., Trinder, P. W., Hammond, K., Junaidu, S. B., Morgan, R. G. and Jones, S. L. Peyton (1999) Engineering parallel symbolic programs in GPH. *Concurrency – Practice & Exper*. **11**(12), 701–752.

Loveman, D. B. (1993) High Performance Fortran. *IEEE J. Parallel & Distributed Tech.: Systems & Applic*. **1**(1), 25–42.

Markatos, E. P. and LeBlanc, T. J. (1994) Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Trans. Parallel & Distributed Syst*. **5**(4), 379–400.

McGraw, J. R., Skedzielewski, S. K., Allan, S. J., Oldehoeft, R. R. *et al.* . (1985) *Sisal: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2*. M 146. Lawrence Livermore National Laboratory, Livermore, California, USA.

Michaelson, G., Scaife, N., Bristow, P. and King, P. (2001) Nested algorithmic skeletons from higher order functions. *Parallel Algorithms & Applic*. **16**, 181–206.

Nöcker, E., Plasmeijer, M. J. and Smetsers, S. (1991) The Parallel ABC-machine. In: Glaser, H. and Hartel, P., editors, *Proceedings 3rd International Workshop on Parallel Implementations of Functional Languages*, pp. 351–382. Southampton, UK. Technical Report CSTR 91-07.

O'Boyle, M. F. P., Kervella, L. and Bodin, F. (1995) Synchronisation minimisation in a SPMD execution model. *J. Parallel & Distributed Comput.*, **29**(2), 196–210.

Oldehoeft, R. R. (1992) Implementing arrays in SISAL 2.0. *Proceedings 2nd Sisal Users Conference*, pp. 209–222. San Diego, CA. Lawrence Livermore National Laboratory.

Oldehoeft, R. R., Cann, D. C. and Allan, S. J. (1986) SISAL: Initial MIMD performance results. In: Händler, W., Haupt, D., Jeltsch, R., Juling, W. and Lange, O., editors, *Proceedings Conference on Algorithms and Hardware for Parallel Processing (CONPAR'86): Lecture Notes in Computer Science 237*, pp. 120–127. Aachen, Germany. Springer-Verlag.

Polychronopoulos, C. and Kuck, D. (1987) Guided Self Scheduling. *IEEE Trans. Comput*. **36**(12), 1425–1439.

Reppy, J. H. (1991) CML: A higher-order concurrent language. *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'91)*, pp. 293–305. Toronto, Canada. ACM Press.

Roth, G. (1997) *Optimizing Fortran90D/HPF for Distributed-Memory Computers*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas, USA.

Scholz, S.-B. (1997) On programming scientific applications in SAC – A functional language extended by a subsystem for high-level array operations. In: Kluge, W., editor, *Proceedings 8th International Workshop on Implementation of Functional Languages (IFL'96): Lecture Notes in Computer Science 1268*, pp. 85–104. Bonnn, germany. Springer-Verlag.

Scholz, S.-B. (1998) With-loop-folding in SAC – Condensing consecutive array operations. In: Clack, C., Davie, T. and Hammond, K., editors, *Proceedings 9th International Workshop on Implementation of Functional Languages (IFL'97): Lecture Notes in Computer Science 1467*, pp. 72–92. Springer-Verlag.

Scholz, S.-B. (1999) A case study: Effects of WITH-Loop folding on the NAS Benchmark MG in SAC. In: Hammond, K., Davie, T. and Clack, C., editors, *Proceedings 10th International Workshop on Implementation of Functional Languages (IFL'98): Lecture Notes in Computer Science 1595*, pp. 216–228. London, UK. Springer-Verlag.

Scholz, S.-B. (2003) Single assignment C – Efficient support for high-level array operations in a functional setting. *J. Funct. Program.* **13**(6), 1005–1059.

Serrarens, P. R. (1997) Implementing the conjugate gradient algorithm in a functional language. In: Kluge, W., editor, *Proceedings 8th International Workshop on Implementation of Functional Languages (IFL'96): Lecture Notes in Computer Science 1268*, pp. 125–140. Springer-Verlag.

Serrarens, P. R. (1999) Explicit message passing for Concurrent Clean. In: Hammond, K., Davie, T. and Clack, C., editors, *Proceedings 10th International Workshop on Implementation of Functional Languages (IFL'98): Lecture Notes in Computer Science 1595*, pp. 229–245. London, UK. Springer-Verlag.

Skedzielewski, S. and Welcome, M. L. (1985) Data flow graph optimization in IF1. In: Jouannaud, J.-P., editor, *Proceedings Conference on Functional Programming Languages and Computer Architecture (FPCA'85): Lecture Notes in Computer Science 201*, pp. 17–34. Nancy, France. Springer-Verlag.

Squillante, M. S. and Lazowska, E. D. (1993) Using processor-cache Affinity information in shared-memory multiprocessor scheduling. *IEEE Trans. Parallel & Distributed Syst.* **4**(2), 131–143.

Tang, P. and Yew, P.-C. (1986) Processor self-scheduling for multiple nested Pparallel loops. *Proceedings International Conference on Parallel Processing (ICPP'86)*, pp. 528–535. Chicago, IL.

Trinder, P., Hammond, K., Loidl, H.-W. and Jones, S. L. Peyton (1998) Algorithm + Strategy = Parallelism. *J. Funct. Program.* **8**(1), 23–60.

Trinder, P. W., Hammond, K., Loidl, H.-W., Jones, S. L. Peyton and Wu, J. (1996) Accidents always come in trees: A case study of data-intensive programs in Parallel Haskell. In: Trinder, P. W., editor, *Proceedings 9th Glasgow Functional Programming Workshop (GFPW'96)*, Ullapool, Scotland.

Trinder, P. W., Jr., Barry, E., Davis, M. K., Hammond, K., Junaidu, S. B., Klusik, U., Loidl, H.-W. and Jones, S. L. Peyton (1999) GPH: An architecture-independent functional language. *IEEE Trans. Softw. Eng.* Submitted.

Tseng, C.-W. (1993) *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, Department of Computer Science, Houston, Texas, USA.

Tseng, C.-W. (1995) Compiler optimizations for eliminating barrier synchronization. *Proceedings 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP'95)*, pp. 144–155. Santa Barbara, CA. ACM Press.

Tzen, T. H. and Ni, L. M. (1993) Trapezoid self-scheduling: A practical scheduling scheme for parallel computers. *IEEE Trans Parallel & Distributed Syst.* **4**(1), 87–98.

van Groningen, J. (1997) The implementation and efficiency of arrays in Clean 1.1. In: Kluge, W., editor, *Proceedings 8th International Workshop on Implementation of Functional Languages (IFL'96): Lecture Notes in Computer Science 1268*, pp. 105–124. Bonn, Germany. Springer-Verlag.

Wadler, P. (1992) The essence of functional programming. *Proceedings 19th ACM Symposium on Principles of Programming Languages (OPOL'92)*, pp. 1–14. Albuquerque, NM. ACM Press.

Wolfe, M. J. (1995) *High-Performance Compilers for Parallel Computing*. Addison-Wesley.

Yan, Y., Jin, C. and Zhang, X. (1997) Adaptively scheduling parallel loops in distributed shared-memory systems. *IEEE Trans. Parallel & Distributed Syst*. **8**(1), 70–81.

Zima, H. P. and Chapman, B. (1991) *Supercompilers for Parallel and Vector Computers*. Addison-Wesley.

Zörner, T. (1998) Numerical analysis and functional programming. In: Hammond, K., Davie, T. and Clack, C., editors, *Proceedings 10th International Workshop on Implementation of Functional Languages (IFL'98)*, pp. 27–48. London,UK.