Optimizations on Array Skeletons in a Shared Memory Environment

Clemens Grelck

Medical University of Lübeck [*]
Institute for Software Technology and Programming Languages
D–23569 Lübeck, Germany
grelck@isp.mu-luebeck.de

Abstract. Map- and fold-like skeletons are a suitable abstractions to guide parallel program execution in functional array processing. However, when it comes to achieving high performance, it turns out that confining compilation efforts to individual skeletons is insufficient. This paper proposes compilation schemes which aim at reducing runtime overhead due to communication and synchronization by embedding multiple array skeletons within a so-called `spmd` meta skeleton. Whereas the meta skeleton exclusively takes responsibility for the organization of parallel program execution, the original array skeletons are focussed to their individual numerical operation. While concrete compilation schemes assume multithreading in a shared memory environment as underlying execution model, ideas can be carried over to other settings straightforwardly. Preliminary performance investigations help to quantify potential benefits.

1 Introduction

Algorithmic skeletons [7] are a well-known framework to express parallel computations on a high level of abstraction. In the field of array processing familiar skeletons like `map`, `zip`, and `fold` are complemented by more array-specific ones like `take`, `drop`, or `rotate`. However, serious array processing generally requires many more such basic building blocks. Dedicated languages, both functional and imperative, usually provide a more or less comprehensive collection.

In a shared memory environment, where explicit data decomposition is not necessary, the organization of parallel program execution can completely be confined to specific implementations of these array skeletons. Whereas programs are still executed sequentially by a single thread of control in general, evaluation of array skeletons causes additional *worker threads* to be created. They simultaneously perform the specified computations in a cooperative manner and terminate after synchronization with the initial or *master thread*. Having waited for the termination of the last worker thread, the master thread continues with sequential operations.

[*] Most of the work described in this paper was done while the author was affiliated with the University of Kiel.

The appealing property of this approach is that it directs all efforts for non-sequential program execution to specific implementations of some high-level language constructs. Unfortunately, it shows some drawbacks as well. Individual compilation and execution of each instance of a skeleton results in a sequence of thread creation, synchronization, and termination operations, all of which are costly with respect to execution time. As a consequence, the runtime overhead associated with non-sequential program execution may severely limit the performance gains actually achieved.

Optimizations which aim at reducing this runtime overhead by combining multiple instances of skeletons into a single one are restricted by the inherent need to represent the combined operation within the given skeletal framework. At this point, it turns out to be an obstacle that array skeletons mix up two different aspects of program execution. On the one hand, they specify some numerical computations to be performed, regardless of whether this is done sequentially or in parallel. On the other hand, they specify a model for organizing parallel program execution. Unfortunately, restrictions with respect to the former aspect usually hinder optimizations concerning the latter one.

As the major contribution we propose transformation schemes which at some intermediate level of compilation explicitly separate the organizational aspects of array skeletons from their computational aspects. Each instance of an array skeleton is embedded within a so-called `spmd` meta skeleton. The name stands for "single program, multiple data". It refers to the fact that this skeleton represents areas of parallel execution within a program which is executed sequentially otherwise. Multiple threads execute the same code but act on pairwise disjoint parts of the arrays involved. We call it a *meta* skeleton because it has no computational meaning for itself; it solely specifies a program's coordination behaviour.

Embedded within `spmd` meta skeletons, variants of the original array skeletons define the computations actually to be performed. They differ from their original counterparts in that most organizational aspects of their parallel execution are stripped off. In fact, they expect to be evaluated in a parallel execution environment which is already set up appropriately (by the `spmd` meta skeleton) rather than doing so themselves.

This two-level representation of parallel array operations forms the basis for optimizations across multiple instances of array skeletons. By merging several `spmd` meta skeletons into a single one, considerable overhead for the organization of parallel program execution can be shared among multiple array skeletons. As a consequence, the ratio between productive computations and organizational overhead is improved. The application of this optimization is only restricted by data dependencies, whereas embedded numerical operations are not concerned.

The paper is organized as follows. Section 2 sketches out implementations of array skeletons in a shared memory environment. Section 3 introduces the `spmd` meta skeleton along with the associated optimization and code generation schemes; Section 4 refines the optimization scheme. Preliminary performance investigations are discussed in Section 5 while Section 6 outlines some related work. Section 7 concludes and discusses directions for future work.

2 Implementing Array Skeletons

Various array skeletons have been proposed in different contexts [1,2]. Most of them fall into one of two basic categories: they either create a new array whose elements are individually computed from some arguments based on their index positions or they perform a reduction operation. To abstract from individual properties of concrete skeletons we introduce two generalized skeletons representing the two basic categories:

$$\texttt{GenArray(}\ shp, op_{idx}(arg_1, \ldots, arg_n)\ \texttt{)} \quad ,$$
$$\texttt{FoldArray(}\ shp, op_{idx}(arg_1, \ldots, arg_n), fold\_op, neutral\ \texttt{)} \quad .$$

The `GenArray` skeleton creates a new array of shape *shp*, where *shp* is considered to denote an integer vector defining both the dimensionality of the new array as well as its extent in each dimension. Explicit specification of the shape vector allows for any relationship between shapes and even values of arguments and the shape of the result array. Its elements are separately computed by some operation op_{idx}. Being parameterized over individual index positions, different operations may actually be realized by op_{idx} on disjoint areas of the result array. With its arity left unspecified any number of scalar and array arguments may occur. So, these skeletons may be considered operational templates rather than concrete higher-order functions.

Similar to `GenArray`, the `FoldArray` skeleton evaluates the given function op_{idx} for each legal index position associated with the shape *shp*. Instead of using the results for initializing a new array, they are pairwise folded using the binary fold operation *fold_op* with neutral element *neutral*. Since the concrete sequence of folding operations is left unspecified, legal fold operations must be associative and commutative to ensure deterministic results.

Fig. 1 shows the compilation scheme for the `GenArray` skeleton into imperative pseudo code both for the master thread (left-hand side) and for worker threads (right-hand side). Whenever the master thread evaluates a `GenArray` skeleton, it first allocates memory for storing the result array, which is referred to by some previously unused variable `tmp`. Due to the commitment to shared memory architectures, no explicit data decomposition is required, and implicit dynamic memory management for arrays can be adopted from sequential implementations with little or no alteration. Afterwards, the base address and the shape of the result array as well as the numerical arguments of the skeleton are broadcast, and, last but not least, the desired number of worker threads is created.

At a first glance, it seems to be inconsistent to send data to worker threads before they actually exist. However, in a shared memory environment `send` and `receive` operations are nothing but copy operations to and from some specific memory buffer, which may exist independently of the threads themselves. Broadcasting data prior to thread creation allows for a non-blocking implementation of the corresponding `receive` operations, and, thus, reduces synchronization requirements among threads to their creation.

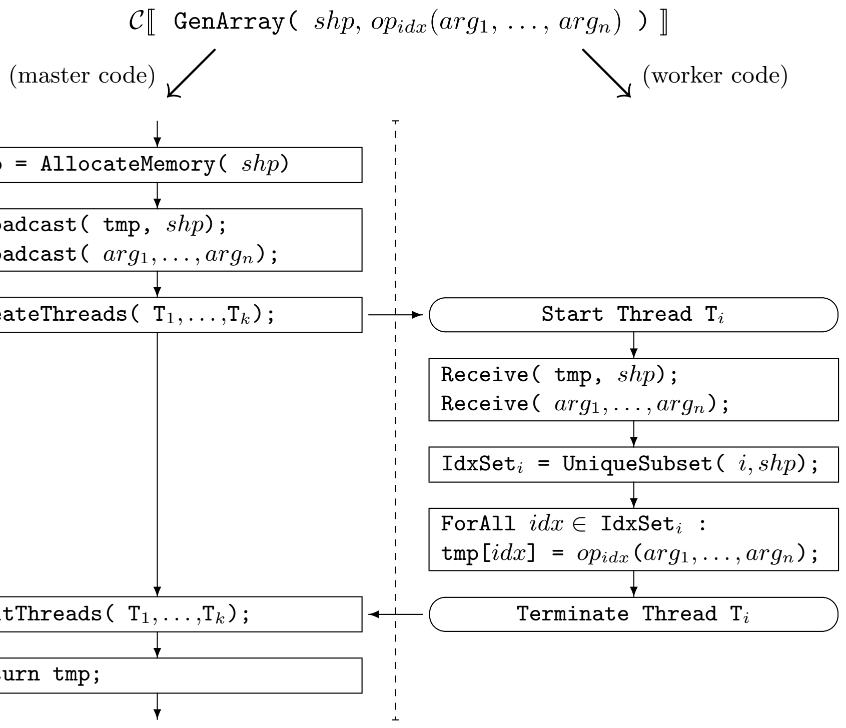
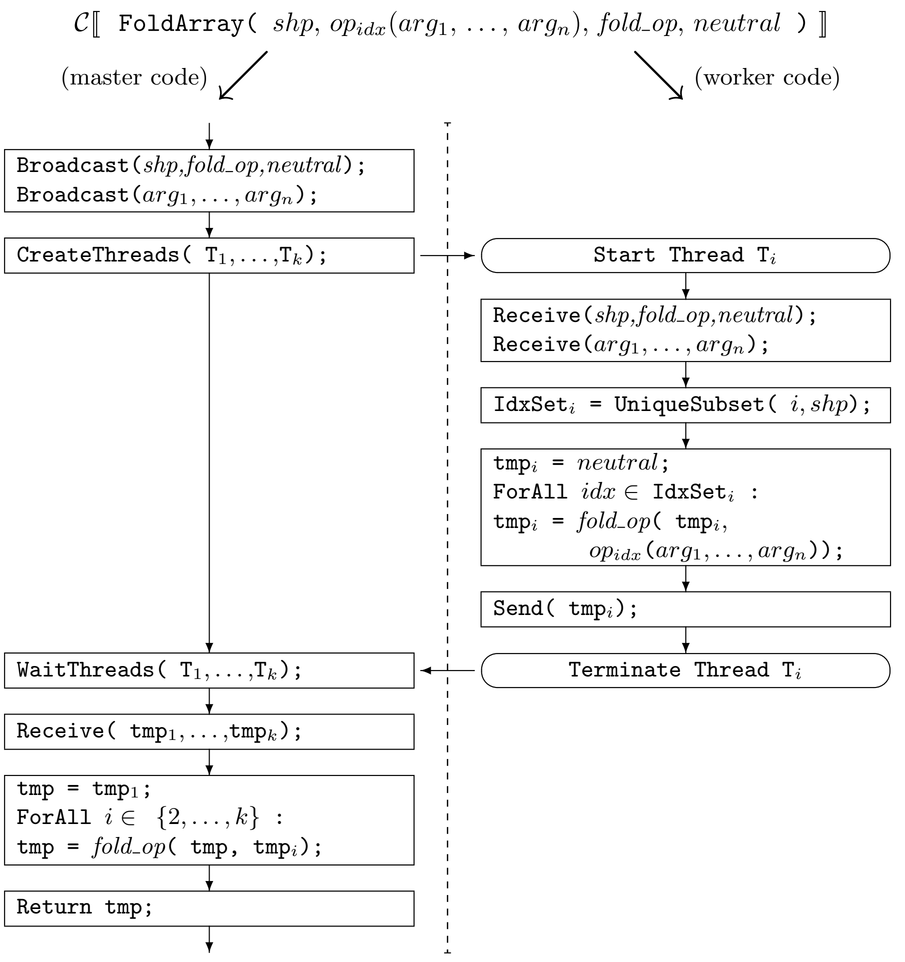

$\mathcal{C}[\![$ GenArray(shp, $op_{idx}(arg_1, \ldots, arg_n)$) $]\!]$
(master code)
(worker code)
tmp = AllocateMemory(shp)
Broadcast(tmp, shp);
Broadcast($arg_1, \ldots, arg_n$);
CreateThreads($T_1, \ldots, T_k$);
WaitThreads($T_1, \ldots, T_k$);
Return tmp;
Start Thread T_i
Receive(tmp, shp);
Receive($arg_1, \ldots, arg_n$);
$IdxSet_i$ = UniqueSubset(i, shp);
ForAll $idx \in IdxSet_i$:
tmp[idx] = $op_{idx}(arg_1, \ldots, arg_n)$;
Terminate Thread T_i

Fig. 1. Compilation scheme for the `GenArray` skeleton.

All worker threads uniformly execute the code shown on the right hand side of Fig. 1, but each thread may identify itself by means of a unique ID. As a first step, a worker thread receives the target array's base address and shape along with the other arguments in order to set up an appropriate execution environment for performing the numerical operation. Then, based on its unique ID, each worker thread identifies a subspace of the entire index space defined by *shp*. Proper implementations of `UniqueSubset` guarantee that each legal index position belongs to exactly one such index subspace. For each element of its individual index subspace a worker thread computes the corresponding numerical operation and initializes the result array accordingly. After having completed their individual computations, the worker threads terminate.

While worker threads are responsible for the parallel execution of the numerical computations represented by the skeleton, the master thread just awaits the termination of all worker threads. As soon as the last worker thread has completed its individual share of work, the master thread returns the result to the surrounding context and continues with sequential program execution until the next skeleton is encountered. Altogether, program execution is organized as a sequence of concurrent stages.



𝒞⟦ FoldArray(shp, op_idx(arg_1, …, arg_n), fold_op, neutral) ⟧
(master code)
(worker code)
Broadcast(shp,fold_op,neutral);
Broadcast(arg_1,...,arg_n);
CreateThreads(T_1,...,T_k);
Start Thread T_i
Receive(shp,fold_op,neutral);
Receive(arg_1,...,arg_n);
IdxSet_i = UniqueSubset(i,shp);
tmp_i = neutral;
ForAll idx ∈ IdxSet_i :
tmp_i = fold_op(tmp_i, op_idx(arg_1,...,arg_n));
Send(tmp_i);
Terminate Thread T_i
WaitThreads(T_1,...,T_k);
Receive(tmp_1,...,tmp_k);
tmp = tmp_1;
ForAll i ∈ {2,...,k} :
tmp = fold_op(tmp, tmp_i);
Return tmp;

Fig. 2. Compilation scheme for the `FoldArray` skeleton.

Multithreaded code generated for the `FoldArray` skeleton, as shown in Fig. 2, is similar to the implementation of `GenArray`. The master thread broadcasts arguments of the skeleton's numerical operation along with the fold operation, its neutral element, and the corresponding shape and, subsequently, creates the worker threads. Having set up its individual execution environment, each worker thread identifies some unique iteration subspace, just as in the `GenArray` case. It then initializes a local accumulation variable $\texttt{tmp}_i$ by the neutral element of the fold operation and then performs the specified computations restricted to the individual index subspace identified before. Hence, each worker thread computes a partial fold result, which it sends back to the master thread prior to its termination.

The master thread awaits the termination of all worker threads before it receives their partial fold results. Once again, necessary thread management operations are exploited to ensure proper synchronization upon `send`/`receive` communication. Finally, the master thread itself combines the various partial fold results to generate the overall result, which thereupon is returned to the surrounding context.

3 SPMD Optimization

Compiling each instance of the two array skeletons individually directly leads to a fork/join execution model, as illustrated in Fig. 3. Program execution repeatedly alternates between two different modes: sequential execution by the master thread and parallel execution by a given number of worker threads. Unfortunately, switching from one mode to the other inflicts significant runtime overhead:

- creation of worker threads,
- communication from master thread to worker threads,
- communication from worker threads to master thread (`FoldArray` only),
- termination of worker threads,
- synchronization of master thread with worker threads.

Since all this overhead is directly related to parallel program execution itself, it may severely reduce potential benefits in terms of reduced program runtimes.

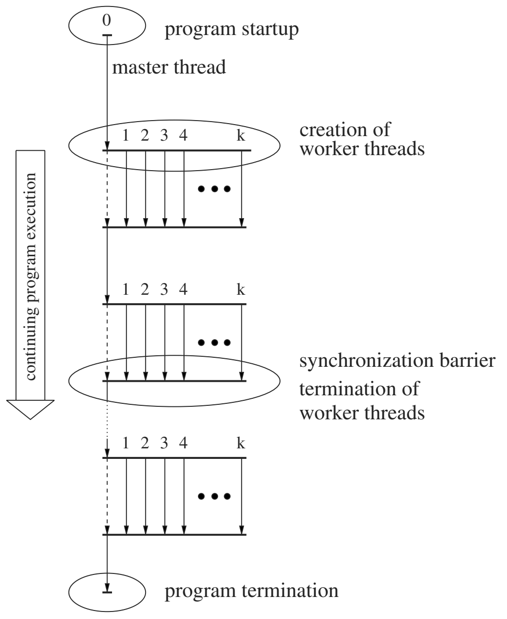


Fig. 3. Illustration of fork/join execution model.

As already pointed out before, array skeletons leave little opportunities for optimization, as they amalgamate directions for parallel program execution with potentially complex numerical operations. To overcome these limitations we adopt a two-level representation of array skeletons. On the outer level the new meta skeleton

`spmd(` *USE, MAP, FOLD, SKEL*`)`

represents the coordination behaviour, whereas — embedded as fourth parameter — variants of the array skeletons define the computations to be performed in parallel. The first three parameters *USE*, *MAP*, and *FOLD* are sets of program identifiers, which serve organizational purposes during subsequent optimization and compilation steps.

Fig. 4 contains the definition of a transformation scheme $\mathcal{SPMD}$, which introduces `spmd` meta skeletons around instances of original array skeletons. For reasons of simplicity, here and in the sequel of this paper it is assumed that code has been transformed into sequences of nested `let`-expressions with a single variable binding and a simple, unnested expression as definition.

To allow for compiling `spmd` skeletons independently from the array skeletons collected in the fourth parameter, additional information is gathered and maintained in the first three parameters. The first parameter *USE* contains the set of argument variables, characterizing the data which have to be broadcast to worker threads when switching to parallel program execution. The *MAP* and *FOLD* sets

```
SPMD ⟦let A = GenArray( shp, op_idx(arg_1,...,arg_n)) in Rest⟧
⟹ let A = spmd( { arg_1,...,arg_n, shp },
                { [ tmp_A, shp ] },
                { },
                let tmp_A = GenArray'( shp, op_idx(arg_1,...,arg_n))
                in [tmp_A] )
    in  SPMD ⟦Rest⟧

SPMD ⟦let A = FoldArray( shp, op_idx(arg_1,...,arg_n), fop, neu) in Rest⟧
⟹ let A = spmd( { arg_1,...,arg_n, shp, fop, neu },
                { },
                { [ tmp_A, fop ] },
                let tmp_A = FoldArray'( shp, op_idx(arg_1,...,arg_n),
                                         fop, neu)
                in [tmp_A] )
    in  SPMD ⟦Rest⟧

SPMD ⟦let A = expr in Rest⟧
⟹ let A = expr in  SPMD ⟦Rest⟧

SPMD ⟦expr⟧
⟹ expr
```

Fig. 4. Embedding array skeletons within `spmd` skeletons.

collect pairs consisting of result variables of `GenArray` and `FoldArray` skeletons and of the associated shape or fold operation, respectively. This information suffices to generate appropriate code from `spmd` skeletons without identifying the individual array skeletons embedded inside. They are marked by a prime indicating that they differ from the original array skeletons in that they expect to be evaluated in a multithreaded execution environment which is already set up appropriately rather than doing so themselves.

```
$\mathcal{MERGE}[\![$ let $A$ = spmd( $USE_A$, $MAP_A$, $FOLD_A$,
                         let $assigns_A$ in $vec_A$)
        in let $B$ = spmd( $USE_B$, $MAP_B$, $FOLD_B$,
                         let $assigns_B$ in $vec_B$)
          in $Rest$ $]\!]$
$\Longrightarrow \mathcal{MERGE}[\![$ let $A$++$B$ = spmd( $USE_A \cup USE_B$,
                                   $MAP_A \cup MAP_B$,
                                   $FOLD_A \cup FOLD_B$,
                                   let $assigns_A$
                                   in let $assigns_B$
                                     in $vec_A$++$vec_B$)
            in $Rest$ $]\!]$
 |   $A \cap USE_B = \emptyset = A \cap B$
$\Longrightarrow$ let $A$ = spmd( $USE_A$, $MAP_A$, $FOLD_A$,
                  let $assigns_A$ in $vec_A$)
    in $\mathcal{MERGE}[\![$ let $B$ = spmd( $USE_B$, $MAP_B$, $FOLD_B$,
                              let $assigns_B$ in $vec_B$)
               in $Rest$ $]\!]$
 |   $OTHERWISE$
```

Fig. 5. SPMD optimization scheme $\mathcal{MERGE}$.

Introducing `spmd` meta skeletons around individual array skeletons itself does not alter the generation of multithreaded code at all. It lays the foundation for a subsequent optimization step, which aims at merging several `spmd` skeletons into a single one. This is formalized by means of the transformation scheme $\mathcal{MERGE}$, shown in Fig. 5. It identifies pairs of `spmd` skeletons which are directly adjacent in the linear nesting of `let`-expressions and which are free of data dependencies. Such pairs are combined into a single `spmd` skeleton which is then characterized by multiple array skeletons and hence multiple return values. An example which illustrates both the introduction of `spmd` skeletons as well as the merging step is given in Fig. 6.

The SPMD optimization may lead to complex `spmd` skeletons containing many different compound array operations. Nevertheless, compilation schemes for individual array skeletons, as outlined in Section 2, can be carried over to the new `spmd` skeleton almost straightforwardly. As shown in Fig. 7, compiling `spmd` skeletons mostly combines elements from both previous compilation schemes.

```
let A = GenArray( shp_A, op_A( D, k, C))
in let B = GenArray( shp_B, op_B( C))
   in let d = FoldArray( shp_d, op_d( D), fun, neu)
      in ...
              ⇓                ⇓                ⇓
let A = spmd( {shp_A, D, k, C},
              {[tmp_A, shp_A]},
              { },
              let tmp_A = GenArray'( shp_A, op_A( D, k, C))
              in [tmp_A])
in let B = spmd( {shp_B, C},
                 {[tmp_B, shp_B]},
                 { },
                 let tmp_B = GenArray'( shp_B, op_B( C))
                 in [tmp_B] )
   in let d = spmd( {shp_d, fun, neu, D},
                    { },
                    {[tmp_d, fun]},
                    let tmp_d = FoldArray'( shp_d, op_d( D),
                                            fun, neu)
                    in [tmp_d] )
      in ...
              ⇓                ⇓                ⇓
let [A,B,d] = spmd( {shp_A, shp_B, shp_d, fun, neu, C, D, k},
                    {[tmp_A, shp_A], [tmp_B, shp_B]},
                    {[tmp_d, fun]},
                    let tmp_A = GenArray'( shp_A, op_A( D, k, C))
                    in let tmp_B = GenArray'( shp_B, op_B( C))
                       in let tmp_d = FoldArray'( shp_d, op_d( D),
                                                  fun, neu)
                          in [tmp_A, tmp_B, tmp_d] )
in ...
```

Fig. 6. Example illustrating SPMD optimization.

The evaluation of an spmd skeleton starts with the allocation of memory for the result arrays of its embedded GenArray skeletons. Both their identifiers as well as their shapes can directly be derived from the MAP set. Then, all variables of the USE and MAP sets are broadcast to the worker threads. After creation, each worker thread immediately sets up its local execution environment for all array skeletons within the spmd skeleton. They are still compiled individually, as indicated by the dashed box. For each skeleton, 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, which eventually folds them using

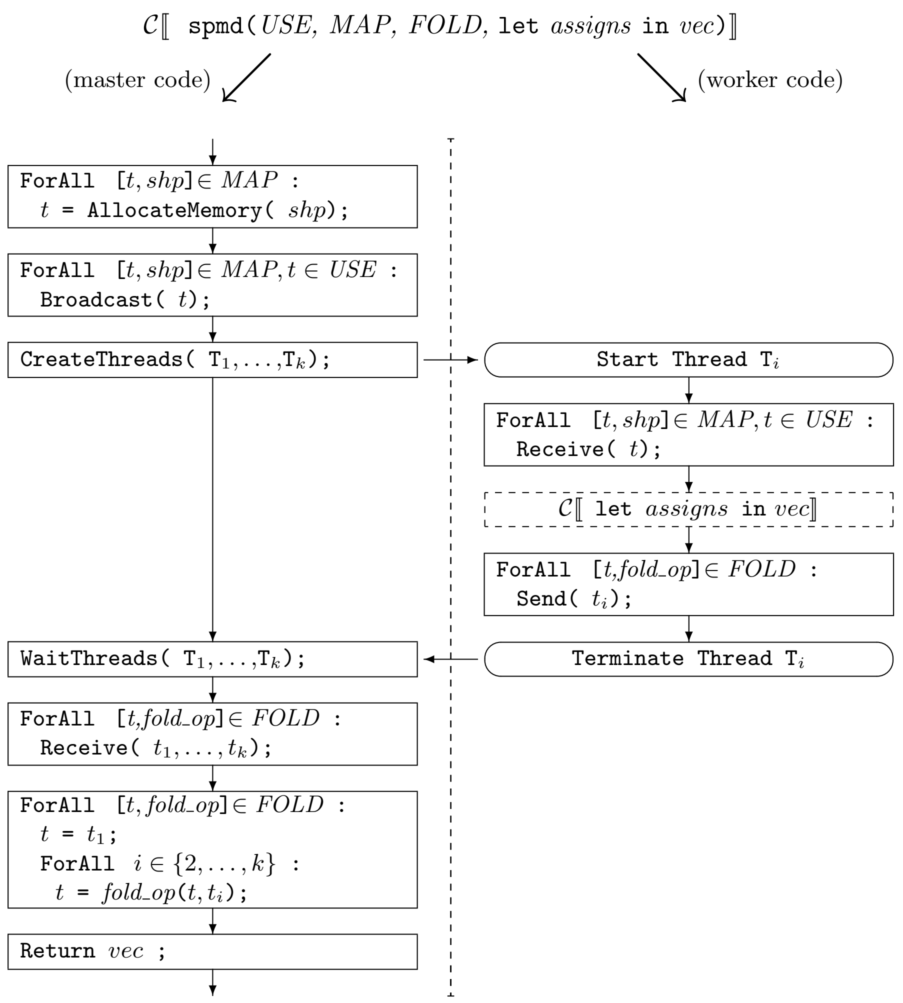

$\mathcal{C}[\![$ spmd(*USE, MAP, FOLD,* let *assigns* in *vec*) $]\!]$
(master code)
(worker code)
ForAll $[t, shp] \in MAP$: t = AllocateMemory(shp);
ForAll $[t, shp] \in MAP, t \in USE$: Broadcast(t);
CreateThreads($T_1, \ldots, T_k$);
Start Thread T_i
ForAll $[t, shp] \in MAP, t \in USE$: Receive(t);
$\mathcal{C}[\![$ let *assigns* in *vec* $]\!]$
ForAll $[t, fold\_op] \in FOLD$: Send(t_i);
Terminate Thread T_i
WaitThreads($T_1, \ldots, T_k$);
ForAll $[t, fold\_op] \in FOLD$: Receive($t_1, \ldots, t_k$);
ForAll $[t, fold\_op] \in FOLD$: t = t_1; ForAll $i \in \{2, \ldots, k\}$: t = $fold\_op(t, t_i)$;
Return *vec* ;

Fig. 7. Compilation scheme for the `spmd` meta skeleton.

the associated fold operations. Finally, the vector of results is returned to the surrounding context.

4 Array Skeletons and Scalar Code

The optimization scheme $\mathcal{MERGE}$, as introduced in the previous section, solely addresses `spmd` skeletons which occur directly adjacent in a linear nesting of `let`-expressions. However, even in programs dominated by array processing, this case is rather an exception. In fact, typical intermediate codes mix up array skeletons with scalar computations. Unfortunately, any scalar `let`-expression in between two skeletal ones prevents their optimization regardless of data dependencies.

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

Fig. 8. Example for restrictions on code reorganization.

In order to make the SPMD optimization more useful in practice, the $\mathcal{MERGE}$ scheme must be extended by a code restructuring component. Scalar `let`-expressions between two consecutive skeletons need to be moved either ahead of the first skeleton or behind the second one. However, the necessary code reorganization is restricted by data dependencies as well as by anti-dependencies. Fig. 8 illustrates some of these restrictions by means of an example. Ignoring the scalar `let`-expressions in between the two `spmd` skeletons makes them perfect candidates for merging. In fact, this is false as there is an indirect data dependency between the two skeletons via the scalar variable `b`. Also the `let`-expression defining `c` may not be moved. Although there are no data dependencies, as in the case of `b`, moving it either ahead of the first or behind the second skeleton would penetrate the binding structure of the entire code fragment.

Fig. 9 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 `let`-expressions and keep track of associated data dependencies and anti-dependencies; application of $\mathcal{MERGE}$ starts with all three being empty.

Leading scalar `let`-expressions are traversed by $\mathcal{MERGE}$ without alteration (3rd rule). The interesting case is encountered when the first skeletal `let`-expression is reached during code traversal. Let us assume, it is followed by a scalar one (2nd rule). If neither data dependencies nor anti-dependencies exist between them, the two `let`-expressions are simply exchanged, thus keeping the chance to merge the `spmd` skeleton with some subsequent one. Otherwise, it may still be possible to push the scalar `let`-expression further down behind a subsequent skeletal `let`-expression. However, whether or not this will be possible with respect to data dependencies or whether or not another `spmd` skeleton follows at all is currently undecidable. Therefore, this decision is postponed by temporarily appending the scalar `let`-expression to the auxiliary store. To keep track of all data dependencies involving such code, two variable sets *use* and *def* are maintained which provide those variables defined or needed by `let`-expressions currently residing in the auxiliary store.

Assuming another scalar `let`-expression follows, it becomes clear that the decision whether or not this can be moved ahead of the preceding skeletal one does not only depend on data dependencies between these two, but also involves all `let`-expressions in the auxiliary store.

```
(1) $\mathcal{MERGE}[\![$ let $A$ = spmd( $USE_A$, $MAP_A$, $FOLD_A$,
                              let $assigns_A$ in $vec_A$)
          in let $B$ = spmd( $USE_B$, $MAP_B$, $FOLD_B$,
                              let $assigns_B$ in $vec_B$)
            in $Rest$ $]\!]$
        $[\![store]\!][\![use]\!][\![def]\!]$

    $\Longrightarrow \mathcal{MERGE}[\![$ let $A$++$B$ = spmd( $USE_A \cup USE_B$,
                                  $MAP_A \cup MAP_B$,
                                  $FOLD_A \cup FOLD_B$,
                                  let $assigns_A$
                                  in let $assigns_B$
                                    in $vec_A$++$vec_B$)
              in $Rest$ $]\!]$
            $[\![store]\!][\![use]\!][\![def]\!]$
      |  $(A \cup def) \cap USE_B = \emptyset = (A \cup def \cup use) \cap B$

    $\Longrightarrow$ let $A$ = spmd( $USE_A$, $MAP_A$, $FOLD_A$,
                     let $assigns_A$ in $vec_A$)
      in $store$
        $\mathcal{MERGE}[\![$ let $B$ = spmd( $USE_B$, $MAP_B$, $FOLD_B$,
                                  let $assigns_B$ in $vec_B$)
                in $Rest$ $]\!]$
              $[\![\ ]\!][\![\ ]\!][\![\ ]\!]$
      |  $OTHERWISE$

(2) $\mathcal{MERGE}[\![$ let $A$ = spmd( $USE_A$, $MAP_A$, $FOLD_A$,
                              let $assigns_A$ in $vec_A$)
          in let $B$ = $expr$
            in $Rest$ $]\!]$
        $[\![store]\!][\![use]\!][\![def]\!]$

    $\Longrightarrow$ let $B$ = $expr$
      in $\mathcal{MERGE}[\![$ let $A$ = spmd( $USE_A$, $MAP_A$, $FOLD_A$,
                                    let $assigns_A$ in $vec_A$)
                in $Rest$ $]\!]$
              $[\![store]\!][\![use]\!][\![def]\!]$
      |  $(A \cup def) \cap USE_{expr} = \emptyset = (A \cup USE_{expr} \cup use) \cap B$

    $\Longrightarrow \mathcal{MERGE}[\![$ let $A$ = spmd( $USE_A$, $MAP_A$, $FOLD_A$,
                                let $assigns_A$ in $vec_A$)
              in $Rest]\!]$
            $[\![store$ let $B$ = $expr$ in $]\!][\![use \cup USE_{expr}]\!][\![def \cup B]\!]$
      |  $OTHERWISE$

(3) $\mathcal{MERGE}[\![$ let $A$ = $expr$ in $Rest$ $]\!]$ $[\![store]\!][\![use]\!][\![def]\!]$
     $\Longrightarrow$ let $A$ = $expr$ in $\mathcal{MERGE}[\![Rest]\!][\![store]\!][\![use]\!][\![def]\!]$

(4) $\mathcal{MERGE}[\![$ $expr$ $]\!][\![store]\!][\![use]\!][\![def]\!]$
     $\Longrightarrow$ $store$ $expr$
```

Fig. 9. Code restructuring SPMD optimization scheme.

Two skeletal let-expressions which directly follow each other or which have been made so by preceding transformations may or may not be merged. Once again this does not only depend on data dependencies between them but also involves the auxiliary store (1st rule). In the absence of dependencies as well as of anti dependencies the two spmd skeletons are merged just as by the initial $\mathcal{MERGE}$ scheme. However, if merging is not possible, all scalar let-expressions from the auxiliary store are re-introduced into the nesting of let-expressions in between the two skeletons and the optimization scheme continues with the second skeletal let-expression. Last but not least, the auxiliary store also needs to be flushed whenever $\mathcal{MERGE}$ reaches the goal expression (4th rule).

Whereas data dependencies reflect the nature of the problem and, therefore, cannot be eliminated, anti-dependencies arise from accidentally giving two different variables the same name. Hence, anti-dependencies may completely be removed by consistent variable renaming. As a consequence, anti dependencies could completely be ignored by $\mathcal{MERGE}$ considerably simplifying its definition. However, such a more far-reaching code transformation is beyond the scope of this paper.

5 Preliminary Performance Investigations

To quantify the effect of the SPMD optimization on the runtime performance of compiled code some experiments have been made in the context of the functional array processing language SAC [23]. Its WITH-loops [15] in some sense represent concrete implementations of the GenArray and FoldArray skeletons discussed throughout this paper. The SAC compiler implicitly derives multi-threaded host machine code for parallel program execution on shared memory multiprocessors using techniques similar to those described in Section 2 [13,14]. The SPMD optimization has not yet been fully implemented, but needed some manual adjustments in intermediate code, namely to realize some of the code reorganization described in Section 4. Test programs were run on a 6-processor SUN Enterprise E4500 as well as on 2 and on 4 identical such systems using a SUN WildFire[1] interconnect [17] to provide a global cache-coherent shared memory with non-uniform memory access times.

The first benchmark code consists of a loop containing a sequence of five GenArray skeletons which each initialize a vector of N integers by some trivial computation. In the absence of data dependencies, the SPMD optimization manages to transform the sequence of individual skeletons into a single spmd meta skeleton and therefore to remove four out of five synchronization barriers in the benchmark code.

[1] "WildFire is Sun's internal code name for an advanced Server architecture that is under development. WildFire prototype systems have not been tested, optimized, or qualified for sale; they have been built for evaluative purposes only. Elements of the WildFire prototype may or may not be used in future Sun products. We obtained the WildFire used in this paper through participation in the WildFire Beta/Collaborative Research program."

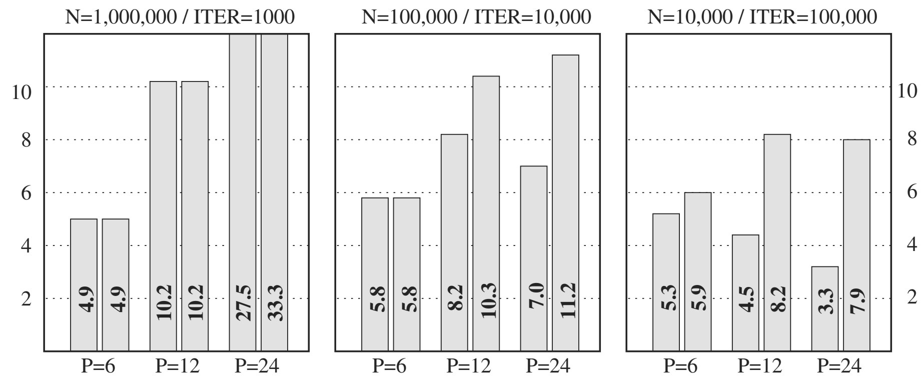


Fig. 10. Speedups with (right bar) and without (left bar) SPMD optimization.

Fig. 10 shows the effect on speedups achieved by parallel program execution for different vector sizes `N`, adjusted numbers of iterations `ITER`, and for the three architectural configurations mentioned before. For each setting the left bar indicates the speedup achieved without SPMD optimization, the right bar gives the speedup with SPMD optimization enabled. Note that between the experiment shown on the left-hand side and that shown on the right-hand side of Fig. 10 the ratio of computations to synchronizations decreases by two orders of magnitude. As expected, the impact of the SPMD optimization on runtime performance grows with program execution increasingly being dominated by synchronization overhead, which in turn grows with increasing processor counts.

However, runtime overhead inflicted by synchronization barriers does not only stem from additional instructions to be executed and from associated com-

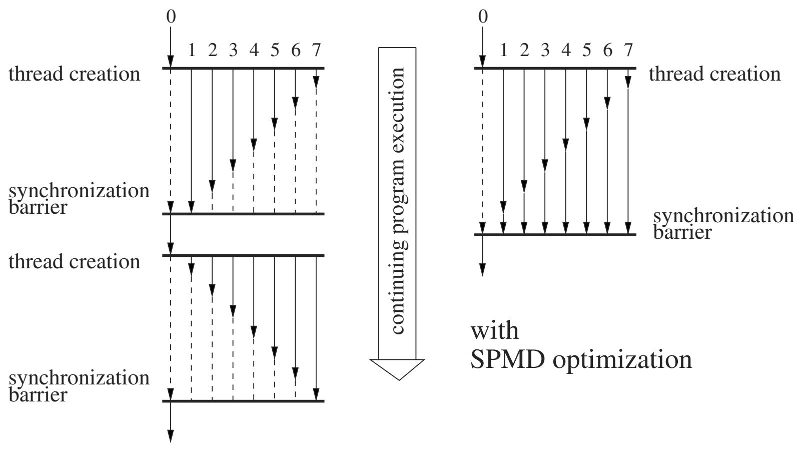


Fig. 11. Load imbalance compensation through SPMD optimization.

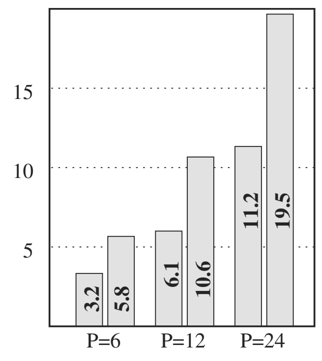


Fig. 12. Performance impact of load imbalance compensation.

munication requirements. Even more significant may be the fact that at each barrier program execution stalls until the very last processor has completed its share of work. Unlike the first experiment, workload distributions among processors are generally imperfect, thus reducing the potential benefits of parallel program execution. Eliminating synchronization barriers by means of the SPMD optimization is likely to result at least in a partial compensation of workload imbalances.

This effect is investigated in the second experiment which maps a complex numerical operation to an upper triangular matrix and, in a subsequent step, to a lower triangular matrix, both of which have 1000^2 elements. Assuming a standard horizontal block decomposition of the operation with respect to the entire matrix, the workload is poorly balanced among the cooperating processors in both individual steps, as illustrated in Fig. 11 for an example using eight processors. However, if the intermediate synchronization barrier can be eliminated, workload imbalances in the two steps are likely to compensate each other almost perfectly. Fig. 12 shows the outcome of this experiment in terms of speedups achieved with and without SPMD optimization. As expected, speedups almost double by SPMD optimization, although the frequency of synchronizations is insignificant due to the large size of the matrices involved.

Though the Sac programs used in the two experiments are rather simple, it is noteworthy that no optimization would have been possible at all by using only the simple $\mathcal{MERGE}$ scheme, introduced in Section 3. Preceding code transformations and optimizations performed by the Sac compiler, in all cases, introduce scalar code in between the array skeletons. This demonstrates the need for the extended optimization scheme introduced in Section 4.

6 Related Work

Research in the area of (parallel) functional programming languages prevailingly has concentrated on algebraic data structures like lists and trees rather than on flat, uniform arrays [18]. Partly due to this more irregular setting, the focus with respect to algorithmic skeletons has been on finding suitable abstractions [22], on expressing applications in terms of skeletons [8], and on constructing skeletons on top of more low-level constructs [25] rather than on optimizing interactions between multiple skeleton instances.

Although not specific to parallel execution, deforestation [9,26] may have a similar effect as the SPMD optimization described in this paper. Whenever explicit construction of intermediate data structures can be avoided, the same holds for associated organizational overhead inflicted by parallel execution. Similar to deforestation in its objective, though different in setting otherwise, WITH-loop-folding [24] condenses consecutive array operations in SAC into single ones. Overhead in case of parallel program execution is condensed incidentally. Nevertheless, both approaches are in some sense orthogonal to SPMD optimization. Whereas they glue together operations that are characterized by the result of one being the argument to the other, the SPMD optimization addresses operations which are unrelated to each other in terms of data dependencies.

With respect to data dependencies, the SPMD optimization is more similar to tupling [5,6]. Yet, once again the setting is very different. Tupling aims at avoiding repeated traversals of the same algebraic data structure by gathering a tuple of results in a single sweep. Similar to deforestation or WITH-loop-folding, tupling may improve the runtime performance of compiled code irrespective of whether it is executed sequentially or in parallel. However, its application is also more restricted making it difficult in general to find suitable candidates for tupling [21]. In contrast, the SPMD optimization solely addresses overhead inflicted by parallel execution. By separating organizational concerns from computational concerns it also needs no assumptions on concrete operations involved.

On the level of programming methodology, frameworks have been developed that combine meaning-preserving transformation rules on a fixed set of skeletons with a cost model guiding their application [10,11]. Code transformations always remain within the given set of user-level skeletons. Similar to deforestation and tupling, their effect in general is not specific to parallel program execution, and their application is also constrained in that concrete operations must meet certain conditions as prerequisites.

In the field of imperative parallel programming, the elimination of synchronization barriers has long been identified as vital for achieving high performance. However, the problem is usually addressed on a lower level of abstraction than in our approach. For instance, code written in the imperative array language ZPL [4] is compiled into sequences of loop nestings and collective data transfer and synchronization operations, so-called *factors* [3]. Subsequent optimizations aim at eliminating superfluous factors and combining ("*joining*") others.

Similar optimizations have been proposed on the level of collective operations in MPI [16]. Certain combinations of adjacent collective operations are replaced by less expensive ones [12].

On an even lower level of abstraction, much research has been carried out on eliminating synchronization barriers in sequences of automatically parallelized FORTRAN-77-style loop nestings [19,20]. These approaches differ from our's not only in the imperative background. Automatic parallelization of loop nestings tends to introduce lots of barriers at first and subsequently applies optimization steps to reduce this number in the presence of dependencies that are often difficult to track. In contrast, high-level compound operations like array skeletons along with preceding transformations on that level already avoid explicit creation of most barriers when solving the same numerical problem. Moreover, the remaining barriers can be addressed with clear knowledge of dependencies.

7 Conclusions and Future Work

Array skeletons are a suitable approach for expressing concurrency in functional array processing both from a specificational as well as from an implementational point of view. Unfortunately, the individual compilation of each instance of an array skeleton generally introduces communication and synchronization overhead which could be avoided. The drawback of array skeletons is that they specify both a (potentially) complex numerical operation as well as the organization of parallel program execution. This often prevents optimizations on the level of array skeletons.

This paper proposes an optimization scheme which explicitly separates both aspects from each other by embedding array skeletons within so-called `spmd` meta skeletons. Whereas the former are mostly restricted to the numerical operation, the latter take responsibility for the organization of parallel execution. Based on a thorough analysis of data dependencies as well as of anti-dependencies, multiple array skeletons may be combined into a single `spmd` meta skeleton. This allows for optimizations on the organizational level regardless of concrete numerical operations involved.

Preliminary performance investigations based on partially hand-optimized multithreaded SAC code reveal considerable improvements whenever the organizational overhead of parallel program execution turns out to be significant. This observation motivates us to fully implement the code restructuring version of the SPMD optimization in the future. More thorough investigations of its performance impact, involving realistic benchmark codes and application programs, have to be undertaken.

Moreover, the SPMD optimization itself may further be improved. Currently, it aggressively seeks to merge array skeletons as far as possible in order to reduce synchronization overhead. Unfortunately, it may considerably increase memory requirements at the same time. Rather than allocating memory for target arrays one after the other in a sequence of `GenArray` skeletons, memory allocation is done for all arrays at once prior to computing their elements. Since it is guaranteed that no data dependencies exist within a merged `spmd` skeleton, this is no problem in itself, as allocated arrays are needed beyond the current context, anyways. However, to the same extent as memory allocations are moved ahead in the code, memory de-allocations are deferred til completion of an entire `spmd` skeleton. Both effects in conjunction may — in pathological cases — increase the overall memory consumption of a program beyond what is available and, hence, hinder a program from running to completion. In order to cope with this effect, the SPMD optimization should further be extended by a component that keeps track of such effects and — whenever necessary — prevents merging of `spmd` skeletons in unfavourable cases.

References

1. J.C. Adams, W.S. Brainerd, J.T. Martin, B.T. Smith, and J.L. Wagener. *Fortran-95 Handbook — Complete ANSI/ISO Reference*. Scientific and Engineering Computation. MIT Press, 1997.
2. G.H. Botorog and H. Kuchen. Efficient High-Level Parallel Programming. *Theoretical Computer Science*, 196(1-2):71–107, 1998.
3. B.L. Chamberlain, S.-E. Choi, E.C. Lewis, C. Lin, L. Snyder, and W.D. Weathersby. Factor-Join: A Unique Approach to Compiling Array Languages for Parallel Machines. In D.C. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D.A. Padua, editors, *Proceedings of the 9th Workshop on Languages and Compilers for Parallel Computing (LCPC'96), San José, California, USA*, volume 1239 of *Lecture Notes in Computer Science*, pages 481–500. Springer-Verlag, 1997.
4. B.L. Chamberlain, S.-E. Choi, E.C. Lewis, C. Lin, L. Snyder, and W.D. Weathersby. The Case for High Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering*, 5(3):76–86, 1998.
5. W. Chin. Towards an Automated Tupling Strategy. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation (PEPM'97), Copenhagen, Denmark*, pages 119–132. ACM Press, 1993.
6. W. Chin. Fusion and Tupling Transformations: Synergies and Conflicts. In *Proceedings of the Fuji International Workshop on Functional and Logic Programming, Susono, Japan*, pages 106–125. World Scientific Publishing, 1995.
7. M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Reserach Monographs in Parallel and Distributed Computing. Pitman, 1989.
8. J. Darlington, A.J. Field, P.G. Harrison, P.H.J. Kelly, D.W.N. Sharp, Q. Wu, and R.L. While. Parallel Programming using Skeleton Functions. In *Proceedings of the Conference on Parallel Architectures and Reduction Languages Europe (PARLE'93)*, volume 694 of *Lecture Notes in Computer Science*, pages 146–160. Springer-Verlag, 1993.
9. A. Gill, J. Launchbury, and S.L. Peyton Jones. A Short Cut to Deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'93), Copenhagen, Denmark*, pages 223–232. ACM Press, 1993.
10. S. Gorlatch and C. Lengauer. (De)Composition Rules for Parallel Scan and Reduction. In *Proceedings of the 3rd International Working Conference on Massively Parallel Programming Models (MPPM'97), London, UK*, pages 23–32. IEEE Computer Society Press, 1997.
11. S. Gorlatch and S. Pelagatti. A Transformational Framework for Skeletal Programs: Overview and Case Study. In J. Rohlim et al., editors, *Parallel and Distributed Processing. IPPS/SPDP'99 Workshops Proceedings*, volume 1586 of *Lecture Notes in Computer Science*, pages 123–137. Springer-Verlag, 1999.
12. S. Gorlatch, C. Wedler, and C. Lengauer. Optimization Rules for Programming with Collective Operations. In M. Atallah, editor, *Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and*

Distributed Processing (IPPS/SPDP'99), San Juan, Puerto Rico, pages 492–499, 1999.
13. C. Grelck. Shared Memory Multiprocessor Support for SAC. In K. Hammond, T. Davie, and C. Clack, editors, *Proceedings of the 10th International Workshop on Implementation of Functional Languages (IFL'98), London, UK, selected papers*, volume 1595 of *Lecture Notes in Computer Science*, pages 38–54. Springer-Verlag, 1999.
14. C. Grelck. *Implicit Shared Memory Multiprocessor Support for the Functional Programming Language SAC — Single Assignment C*. PhD thesis, University of Kiel, Kiel, Germany, 2001. Logos Verlag, Berlin, 2001.
15. C. Grelck, D. Kreye, and S.-B. Scholz. On Code Generation for Multi-Generator WITH-Loops in SAC. In P. Koopman and C. Clack, editors, *Proceedings of the 11th International Workshop on Implementation of Functional Languages (IFL'99), Lochem, The Netherlands, selected papers*, volume 1868 of *Lecture Notes in Computer Science*, pages 77–94. Springer-Verlag, 2000.
16. W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, Massachusetts, USA, 1994.
17. E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th International Conference on High-Performance Computer Architecture (HPCA'99), Orlando, Florida, USA*, pages 172–181. IEEE Computer Society Press, 1999.
18. K. Hammond and G. Michaelson. *Research Directions in Parallel Functional Programming*. Springer-Verlag, 1999.
19. H. Han, C.-W. Tseng, and P. Keleher. Eliminating Barrier Synchronization for Compiler-Parallelized Codes on Software DSMs. *International Journal of Parallel Programming*, 26(5):591–612, 1998.
20. M.F.P. O'Boyle (HP), L. Kervella (HP), and F. Bodin. Sronisation Mininimisation in a SPMD Execution Mode. *Journal of Parallel and Distributed Computing*, 29(2):196–210, 1995.
21. Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling Calculation Eliminates Multiple Data Traversals. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'97), Amsterdam, The Netherlands*. ACM Press, 1997.
22. F.A. Rabhi. Exploiting Parallelism in Functional Languages: A "Paradigm-Oriented“ Approach. In T. Lake and P. Dew, editors, *Abstract Machine Models for Highly Parallel Computers*. Oxford University Press, 1993.
23. S.-B. Scholz. On Defining Application-Specific High-Level Array Operations by Means of Shape-Invariant Programming Facilities. In S. Picchi and M. Micocci, editors, *Proceedings of the International Conference on Array Processing Languages (APL'98), Rome, Italy*, pages 40–45. ACM Press, 1998.
24. S.-B. Scholz. With-loop-folding in SAC — Condensing Consecutive Array Operations. In C. Clack, K.Hammond, and T. Davie, editors, *Proceedings of the 9th International Workshop on Implementation of Functional Languages (IFL'97), St. Andrews, Scotland, UK, selected papers*, volume 1467 of *Lecture Notes in Computer Science*, pages 72–92. Springer-Verlag, 1998.
25. P. Trinder, K. Hammond, H.-W. Loidl, and S.L. Peyton Jones. Algorithm + Strategy = Parallelism. *Journal of Functional Programming*, 8(1):23–60, 1998.
26. P. Wadler. Deforestation: Transforming Programs to Eliminate Trees. *Theoretical Computer Science*, 73(2):231–248, 1990.