# Implementing the NAS Benchmark MG in SAC

Clemens Grelck
Medical University of Lübeck
Institute for Software Technology and Programming Languages
23569 Lübeck, Germany
grelck@isp.mu-luebeck.de

## Abstract

*SAC is a purely functional array processing language designed with numerical applications in mind. It supports generic, high-level program specifications in the style of APL. However, rather than providing a fixed set of built-in array operations, SAC provides means to specify such operations in the language itself in a way that still allows their application to arrays of any dimension and size. This paper illustrates the specificational benefits of this approach by means of a high-level SAC implementation of the NAS benchmark MG realizing 3-dimensional multigrid relaxation with periodic boundary conditions.*

*Despite the high-level approach, experiments show that by means of aggressive compiler optimizations SAC manages to achieve performance characteristics in the range of low-level Fortran and C implementations. For benchmark size class A, SAC is outperformed by the serial Fortran-77 reference implementation of the benchmark by only 23%, whereas SAC itself outperforms a C implementation by the same figure. Furthermore, implicit parallelization of the SAC code for shared memory multiprocessors achieves a speedup of 7.6 with 10 processors. With these figures, SAC outperforms both automatic parallelization of the serial Fortran-77 reference implementation as well as an OpenMP solution based on C code.*

## 1 Introduction

SAC (for Single Assignment C) [26, 27] is a purely functional programming language with a C-like syntax and extended support for $n$-dimensional arrays. It allows for high-level array processing in a way similar to APL [20]. Implicit memory management and the ability to use arrays as function arguments and results without restrictions make array processing in SAC as simple as dealing with scalars in conventional languages.

Unlike APL and many other array languages, e.g. Fortran-90/95 [1], SAC does not provide compound array operations as built-in primitives. Instead, so-called WITH-*loop expressions* (or WITH-loops for short) allow to define such array operations in SAC itself. Still, they may be applied to arrays of any dimension and size, a property which in other languages is usually restricted to built-in primitives. In fact, most operations typical for array languages can be defined as functions in SAC without loss of generality [15].

The advantages of this design are manifold. The core language remains clear and simple; efforts for optimization and parallelization can be focussed on a single though powerful language construct. Any functionality which may be expected from an array processing language is provided through a comprehensive SAC-implemented array library. In contrast to a rich collection of built-in primitives, this organization of the language's array support is much easier to maintain and to extend; portability to different architectures comes for free. Furthermore, programmers themselves can customize the array support to their individual needs just as language implementors. This ability allows for a very generic programming style where application programs are constructed in multiple layers of abstractions based on prespecified building blocks of varying complexity.

The paper demonstrates the practical applicability of this high-level generic programming style by means of a case study. From the popular NAS benchmark suite [2, 4] we have chosen the application kernel MG, which realizes 3-dimensional multigrid relaxation with periodic boundary conditions. Based on the SAC array library it can be implemented by a surprisingly short program, which, to a large extent, turns out to be an almost literal translation of the benchmark's mathematical specification.

Benefits of generic, high-level programming in development and maintenance are only as convincing as the penalty in terms of runtime performance remains acceptable. Therefore, we investigate the performance penalty to be paid by SAC compared with low-level implementations of the benchmark. They are the serial Fortran-77 reference

```
WithLoopExpr  ⇒  with ( Generator ) Operation
Generator     ⇒  Expr Relop Identifier Relop Expr [ Filter ]
Relop         ⇒  < | <=
Filter        ⇒  step Expr [ width Expr ]
Operation     ⇒  genarray ( Expr , Expr )
              |  modarray ( Expr , Expr )
              |  fold ( FoldOp , Expr , Expr )
```

**Figure 1. Syntax of with-loop expressions.**

implementation coming with the NAS benchmark suite and a C implementation directly derived from that code. Despite the high-level approach, experiments show that SAC manages to achieve performance characteristics in the same range as these low-level implementations.

Moreover, the SAC compiler may generate multithreaded code for parallel execution on shared memory multiprocessor systems without any additional programming effort [13, 14]. Experiments with up to 10 processor yield a maximum speedup of 7.6 for SAC, a scaling behaviour that outperforms both automatic parallelization of the serial Fortran-77 reference implementation as well as an OpenMP [9] solution based on the ported C code.

The paper is organized as follows. Section 2 provides a brief introduction to SAC. The NAS benchmark MG is outlined in Section 3, its high-level SAC implementation in Section 4. Section 5 describes the runtime performance experiments in more detail and analyses their outcomes. Some related work is sketched out in Section 6 while Section 7 concludes.

## 2 SAC

The core language of SAC is a functional subset of C, a design which aims at simplifying adaptation for programmers with a background in imperative programming techniques. This kernel is extended by $n$-dimensional arrays as first class objects. SAC provides a small set of built-in array operations, basically primitives to retrieve data pertaining to the structure and contents of arrays, e.g. an array's dimension (`dim(`*array*`)`), its shape (`shape(`*array*`)`), or individual elements (*array*`[`*index-vector*`]`).

Compound array operations are specified using WITH-loop expressions, whose syntax is outlined in Fig. 1. A WITH-loop basically consists of two parts: a *generator* and an *operation*. The generator defines a set of index vectors along with an index variable representing elements of this set. Two expressions, which must evaluate to vectors of equal length, define lower and upper bounds of a rectangular index vector range. An optional filter may further restrict this selection to grids of arbitrary width. Let $a$, $b$, $s$, and $w$ denote expressions that evaluate to vectors of length $n$, then

`(` $a$ `<= i_vec <` $b$ `step` $s$ `width` $w$ `)`

defines the following set of index vectors:

$$\{i\_vec \mid \forall_{j\in\{0,\ldots,n-1\}} : a_j \leq i\_vec_j < b_j \\ \wedge\ (i\_vec_j - a_j) \bmod s_j < w_j\}.$$

The operation specifies the computation to be performed for each element of the index vector set defined by the generator. Let $shp$ denote a SAC-expression that evaluates to a vector, let $i\_vec$ denote the index variable defined by the generator, let $array$ denote a SAC-expression that evaluates to an array, and let $expr$ denote any SAC-expression. Moreover, let $fold\_op$ be the name of a binary commutative and associative function with neutral element $neutral$. Then

- `genarray(` $shp$`,` $expr$`)` generates an array of shape $shp$ whose elements are the values of $expr$ for all index vectors from the specified set, and 0 otherwise;
- `modarray(` $array$`,` $expr$`)` defines an array of shape `shape(`$array$`)` whose elements are the values of $expr$ for all index vectors from the specified set, and the values of $array$`[`$i\_vec$`]` at all other index positions;
- `fold(` $fold\_op$`,` $neutral$`,` $expr$`)` specifies a reduction operation; starting out with $neutral$, the value of $expr$ is computed for each index vector from the specified set and these are subsequently folded using $fold\_op$.

The usage of vectors in WITH-loop generators as well as in the selection of array elements along with the ability to define functions which are applicable to arrays of any dimension and size allows for implementing APL-like compound array operations in SAC itself. This feature is exploited by the SAC array library, which provides, among others, element-wise extensions of arithmetic and relational operators, typical reduction operations like sum and product, various subarray selection facilities, as well as shift and rotate operations. More information on SAC is available at `http://www.informatik.uni-kiel.de/~sacbase/`.

```
Initial solution:        $u = 0$

Each iteration:          $r = v - A\,u$                   (evaluate residual)
                         $u = u + M^k\,r$                 (apply correction)

V-cycle operator $M^k$:  $z_k = M^k\,r_k$ :
    if $k > 1$
        $r_{k-1} = P\,r_k$                    (restrict residual)
        $z_{k-1} = M^{k-1}\,r_{k-1}$          (recursive solve)
        $z_k = Q\,z_{k-1}$                    (prolongate)
        $r_k = r_k - A\,z_k$                  (evaluate residual)
        $z_k = z_k + S\,r_k$                  (apply smoother)
    else
        $z_1 = S\,r_1$                        (apply smoother)
```

**Figure 2. Mathematical specification of NAS-MG given in [3].**

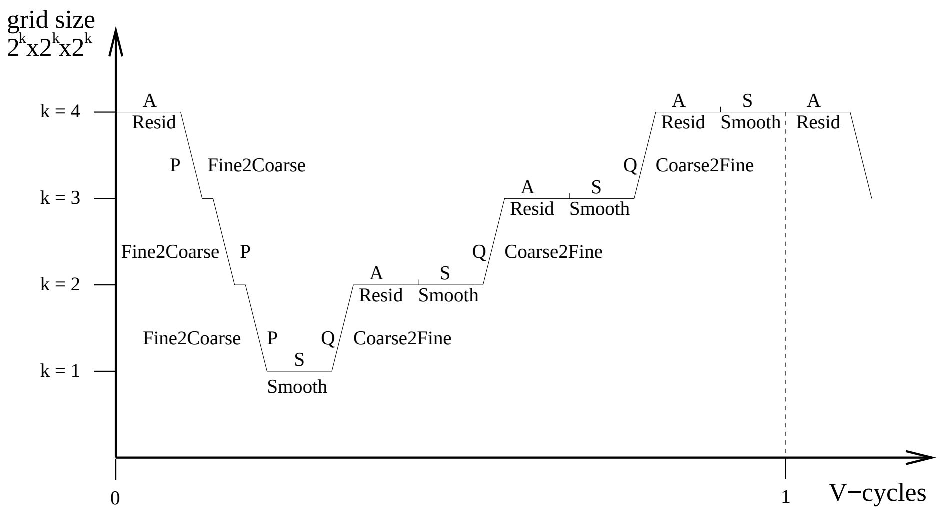


**Figure 3. Illustration of multigrid V-cycle.**

## 3 NAS benchmark MG

The NAS benchmark suite [2, 3, 4] has been developed at NASA Ames Research Center as part of the Numerical Aerodynamic Simulation Program. Its eight benchmarks are considered representative for large scale applications in computational fluid- and aerodynamics.

The application kernel MG implements a V-cycle multigrid algorithm [18, 17] to approximate a solution $u$ of the discrete Poisson equation $\nabla^2 u = v$ on a 3-dimensional grid with periodic boundary conditions. Fig. 2 shows the mathematical specification, as given in [3]. Starting out with the constant zero array as initial solution $u$, each iteration step consists of computing the current residual and, afterwards, applying a correction defined by the recursive V-cycle operator $M^k$, $k = log_2 n$ with $n^3$ being the initial grid size. Relaxation and smoothing steps are recursively embedded within operations to coarsen and refine grid granularities. $A, P, Q$, and $S$ denote 27-point stencil operators whose coefficients are provided by the benchmark specification.

Fig. 3 illustrates the V-cycle algorithm for an initial grid of $16^3$ elements. The order in which the different transformations are applied is depicted along the horizontal axis, whereas the grid size involved in each step is indicated along the vertical axis. After computing an initial residual, the grid is recursively coarsened until it consists of no more than two elements in each dimension. A single smoothing step is applied to the coarsest grid before it is refined again with residual computations and smoothing steps after each refinement until the original granularity is reached.

The rationale behind this multigrid approach is to accelerate the propagation of low-frequent defects across the original grid and, hence, to improve the overall convergence behaviour.

## 4 Implementation in SAC

Following a top-down approach, the mathematical specification of the multigrid V-cycle algorithm, as given in Fig. 2, can almost literally be translated into SAC code. Definitions of the corresponding SAC functions `MGrid` and `VCycle` are shown in Fig. 4. `MGrid` creates the initial solution grid `u` before it alternatingly computes the current residual and the correction of the current solution by means of a complete V-cycle. The recursive function `VCycle` realizes exactly the sequence of basic operations, as illustrated in Fig. 3. Both functions are applicable to arrays of any dimension and size, as indicated by the SAC array type `double[+]`. Although NAS-MG specifically addresses 3-dimensional grids only, this SAC code could be reused for grids of any dimension without alteration.

```
double[+] MGrid( double[+] v,
                 int iter)
{
  u = genarray( shape(v), 0.0);

  for( i=0; i<iter; i+=1)
  {
    r = v - Resid( u);
    u = u + VCycle( r);
  }

  return( u);
}

double[+] VCycle( double[+] r)
{
  if (shape(r)[[0]] > 2 + 2)
  {
    rn = Fine2Coarse( r);
    zn = VCycle( rn);
    z  = Coarse2Fine( zn);
    r  = r - Resid( z);
    z  = z + Smooth( r);
  }
  else
  {
    z  = Smooth( r);
  }

  return( z);
}
```

**Figure 4. Multigrid V-cycle in SAC.**

The arithmetic array operations used in the definitions of `MGrid` and `VCycle` are simply imported from the SAC array library. Additional effort is required to realize the application-specific functions `Resid`, `Smooth`, `Fine2Coarse`, and `Coarse2Fine`. However, any of these operations basically consists of a 27-point stencil relaxation operation with periodic boundary conditions but varying stencil coefficients.

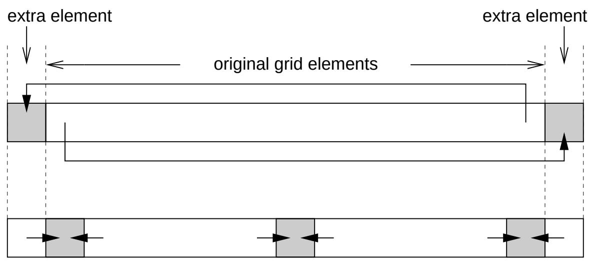


**Figure 5. Adding artificial boundary elements.**

A standard technique for implementing relaxation with periodic boundary conditions is to artificially extend the original grid by additional boundary elements. More precisely, each original boundary element is replicated on the opposite side of the grid, as illustrated in Fig. 5 for a simple vector. This extension allows to implement relaxation with periodic boundary conditions by two consecutive steps. First, artificial boundary elements are initialized according to their location. Afterwards, an ordinary fixed boundary relaxation step is applied to the extended grid. However, extending each array involved by two elements in each dimension requires to adjust the V-cycle termination condition, as can be observed in Fig. 4.

```
double[+] Resid( double[+] u)
{
  u = SetupPeriodicBorder( u);
  u = RelaxKernel( u, A);

  return( u);
}

double[+] Smooth( double[+] r)
{
  r = SetupPeriodicBorder( r);
  r = RelaxKernel( r, S);

  return( r);
}
```

**Figure 6. Implementations of V-cycle functions Resid and Smooth.**

Based on the existing implementation of a relaxation kernel with fixed boundary conditions described in [16], the four V-cycle operations can be implemented straightforwardly. As shown in Fig. 6, implementations of `Resid` and `Smooth` boil down to setting up the periodic boundary elements followed by a single relaxation step. In fact, `Resid`

```
double[+] Fine2Coarse( double[+] r)
{
  rs = SetupPeriodicBorder( r);
  rr = RelaxKernel( r, P);
  rc = condense( 2, rr);
  rn = embed( shape(rc)+1,
              0*shape(rc), rc);

  return( rn);
}

double[+] Coarse2Fine( double[+] rn)
{
  rp = SetupPeriodicBorder( rn);
  rs = scatter(2, rp);
  rt = take( shape(rs)-2, rs);
  r  = RelaxKernel( rt, Q);

  return( r);
}
```

**Figure 7. V-cycle mapping functions.**

and `Smooth` only differ with respect to stencil coefficients defined by the constant vectors `A` and `S`, which are provided by the benchmark specification.

Implementations of the remaining V-cycle functions `Fine2Coarse` and `Coarse2Fine` are shown in Fig. 7. Besides applying additional relaxation steps similar to `Resid` and `Smooth`, but with yet different stencil coefficient vectors `P` and `Q`, they map fine grids to coarse grids and vice versa. These mappings are illustrated in Fig. 8 and in Fig. 9, respectively. Both steps can be realized by combinations of predefined operations from the SAC array library.

The fine-to-coarse mapping is basically implemented by means of the function `condense(`*`str`*`, `*`a`*`)`, which creates an array whose extent in each dimension is by a factor

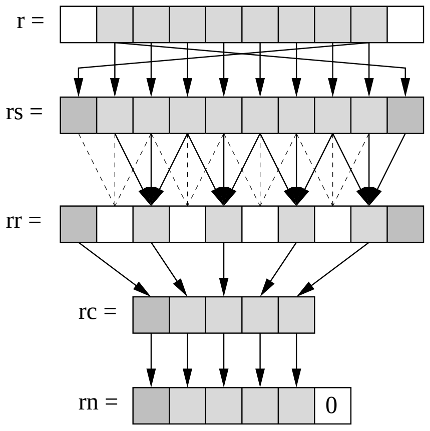


**Figure 8. Fine-to-coarse mapping.**

of *`str`* less than that of the given array *`a`*. Its elements are taken from *`a`* applying a stride of *`str`*. However, due to the additional boundary elements, `condense` does not exactly match the fine-to-coarse mapping requirements, as the result array lacks one such element (see Fig. 8). Therefore, the intermediate array `rc` must be embedded into an array of correct size. This is done by a subsequent application of the library function `embed( `*`shp`*`, `*`pos`*`, `*`a`*`)`, which creates a new array of shape *`shp`*, whose elements — starting at index position *`pos`* — are taken from the argument array *`a`*.

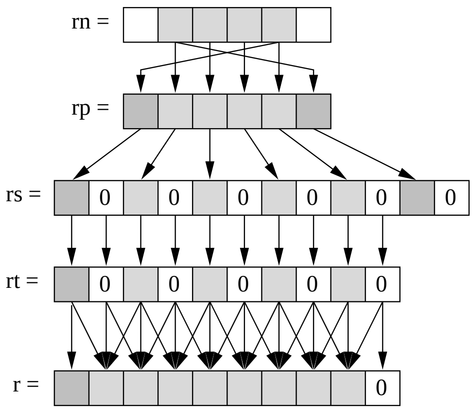


**Figure 9. Coarse-to-fine mapping.**

The coarse-to-fine mapping is implemented by means of the library function `scatter(`*`str`*`, `*`a`*`)`, which is complementary to `condense`. As illustrated in Fig. 9, it creates an array which in each dimension is by a factor of *`str`* larger than the given array *`a`*. All elements of *`a`* are copied into every other location of the result array applying the given stride *`str`*; remaining elements are initialized by zeros. However, due to the additional boundary elements, the resulting array is by two elements too large in each dimension. Subsequent application of the library function `take` removes them.

All array library functions used throughout this section are realized straightforwardly by means of WITH-loop expressions. As shown in Fig. 10, their implementations additionally benefit from some syntactical simplifications which have not yet been addressed. Whenever the required length for vectors in generators of WITH-loop expressions is already determined by the dimension of the result array, simple scalars may be used instead of vectors. They are implicitly replicated to appropriately sized vectors. Moreover, specifications of lower and upper bounds of index vector ranges may be replaced by simple dots denoting the smallest or the largest legal index vector with respect to the shape of the result array. This simplifies the specification of operations which are uniform on all or all inner array elements.

```
double[+] genarray( int[.] shp,
                    double val)
{
  a = with (. <= iv <= .)
      genarray( shp, val);
  return( a);
}

double[+] condense( int str,
                    double[+] a)
{
  ac = with (. <= iv <= .)
       genarray( shape(a) / str,
                 a[str*iv]);
  return( ac);
}

double[+] scatter( int str,
                   double[+] a)
{
  as = with (. <= iv <= . step str)
       genarray( str * shape(a),
                 a[iv/str]);
  return( as);
}

double[+] embed( int[.] shp,
                 int[.] pos,
                 double[+] a)
{
  ae = with (pos <= iv < shape(a) + pos)
       genarray( shp, a[iv-pos]);
  return( ae);
}

double[+] take( int[.] shp,
                double[+] a)
{
  at = with (. <= iv <= .)
       genarray( shp, a[iv]);
  return( at);
}
```

**Figure 10. Array library functions.**

## 5 Experimental Evaluation

This section analyses the runtime performance achieved by code compiled from the SAC specification of NAS-MG, as outlined in the previous section, and compares it with that of the serial Fortran-77 reference implementation as well as with that of a C-based OpenMP solution.

The following experiments were made on a 12-processor SUN Ultra Enterprise 4000 shared memory multiprocessor, running SOLARIS-7. The serial Fortran-77 reference implementation coming with version 2.3 of the NAS benchmark suite was compiled by the SUN Workshop compiler `f77` v5.0; its automatic parallelization feature additionally produced multithreaded code. The OpenMP implementation was compiled by the Omni compiler v1.4a [25, 22] developed by Real World Computing Partnership (RWCP). The code itself has directly been ported by RWCP from the serial Fortran-77 reference implementation to C and afterwards has been decorated with OpenMP directives. [1] Last but not least, SAC code was compiled using the current research compiler `sac2c` v0.91. The SUN Workshop compiler `cc` v5.0 served as a backend compiler for both SAC and OpenMP. Conforming with the benchmark rules, timing was restricted to multigrid iterations and, thus, ignores startup and finalization overhead. Several size classes are defined by the benchmark specification, two of which were selected for the experiments:

- Class W: initial grid size $64^3$ and 40 iterations,
- Class A: initial grid size $256^3$ and 4 iterations.

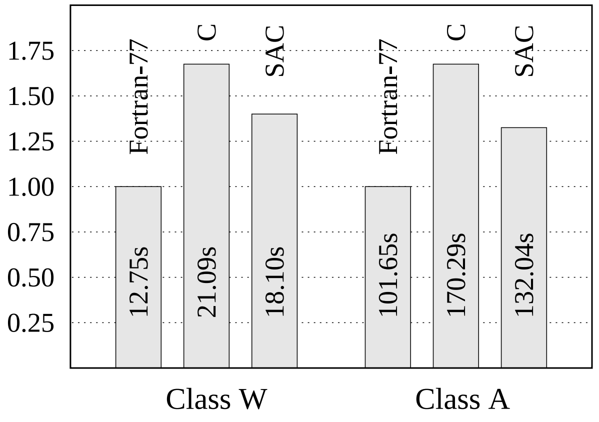
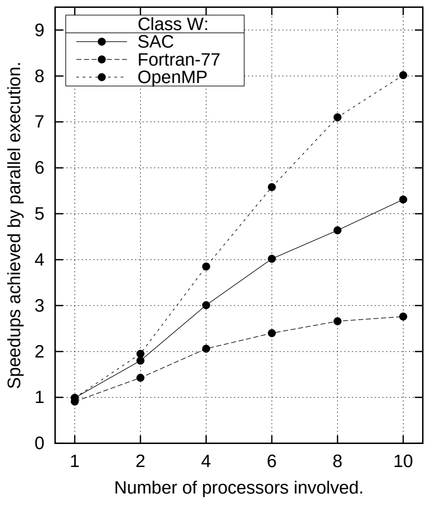


**Figure 11. Single processor performance.**

Fig. 11 shows the runtime performance achieved by all three candidates for both size classes when explicitly being compiled for sequential execution. In fact, the Fortran-77 program outperforms the compiled SAC code by 29.6% and by 23.0% for size classes W and A, respectively, whereas the SAC code in turn outperforms the OpenMP, more precisely C, implementation by 14.2% and by 22.5%. Despite its considerably higher level of abstraction, the SAC specification achieves runtime performance characteristics which are in the same range as the rather well-tuned low-level Fortran-77 and C implementations. Moreover, the runtime performance achieved by SAC improves with increasing problem size, whereas the ratio between C and Fortran-77 seems to be independent of the problem size.

[1]Both the Omni OpenMP compiler as well as the OpenMP implementation of the NAS benchmark suite are available at `http://phase.etl.go.jp/Omni/`.



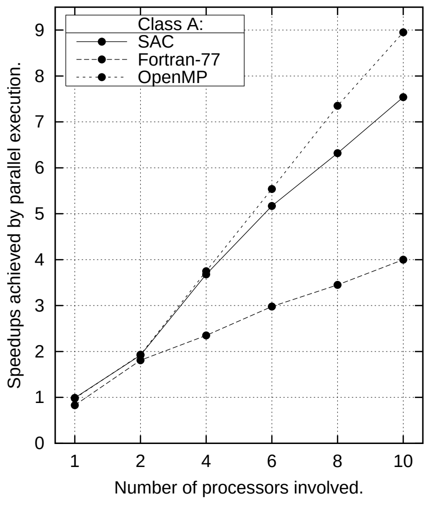


**Figure 12. Speedups relative to individual sequential performance.**

A closer look at the Fortran-77 reference implementation reveals that to some extent tricky hand optimizations are responsible for the superior runtime performance. In general, a 27-point stencil operation incurs 27 multiplications and 26 additions per array element. However, the number of multiplications may be reduced to only four by taking into account that actually just four different coefficients occur in any of the stencil operations. This optimization is realized both explicitly in the Fortran and C/OpenMP implementations as well as by implicit compiler optimization in the case of SAC. However, the low-level codes additionally store intermediate results shared among re-computations of different array elements in auxiliary buffers and thus reduce the actual number of additions to values between 12 and 20 depending on concrete stencils. Unfortunately, the SAC compiler currently does not incorporate appropriate optimization techniques to mimic this low-level implementation trick. Still, it is unclear at the time being why the C/OpenMPimplementation is so much slower than the Fortran-77 reference implementation, although it is almost literally derived from that code. In particular, the same stencil optimization is applied.

Fig. 12 shows the parallel performance achieved by Fortran-77, OpenMP, and SAC using up to ten processors. Simultaneous usage of all twelve processors available was not feasible as the machine is not operated in batch mode and, hence, other system and user processes are always active. Both the Fortran-77 and the SAC implementations are implicitly parallelized without any additional hints to the compilation systems. In the case of OpenMP, a total of 30 manually introduced compilation directives guide the compiler during the parallelization process. All figures are given relative to best individual sequential runtimes. However, the overhead generated by parallel code executed on a single processor is rather small in all three cases.

SAC achieves speedups of up to 5.3 and up to 7.6 for size classes W and A, respectively. Unsurprisingly, the larger problem size A scales much better than size class W. In fact, size class A is the smallest one actually intended for benchmarking, whereas size class W is specifically designed for program development on uniprocessor PCs and workstations [4]. Whereas the scaling behaviour of the automatically parallelized Fortran-77 code is significantly worse than that of SAC reaching speedups of only 2.8 and 4.0, the compiler directive based approach of OpenMP shows the best scalability in the field, leading up to excellent 8.0 and 9.0 for size classes W and A, respectively.

However, sequential base runtimes are as important for parallel performance as scalability. Therefore Fig. 13 shows speedups achieved by all three candidates relative to the fastest sequential solution in the field, i.e. the Fortran-77 reference implementation. With its superior scalability the SAC implementation outperforms the automatically parallelized Fortran-77 code using only four processors. For size class A, superior sequential base performance even allows SAC to stay ahead of OpenMP, at least within the processor range investigated.

In the case of SAC, The main scalability limitation arises from the repeated reduction of the grid size during the V-cycle. Whereas parallel execution pays for larger grids on the top end of the V-cycle, runtime overhead increasingly reduces benefits for smaller grids on the bottom end. Below a certain threshold grid size, it is advised to perform all operations sequentially to avoid excessive overhead. This sequential kernel of the NAS-MG benchmark limits the overall scalability. Although, this problem is algorithm-inherent rather than implementation-specific, its actual impact on runtime performance is significantly increased by dynamic

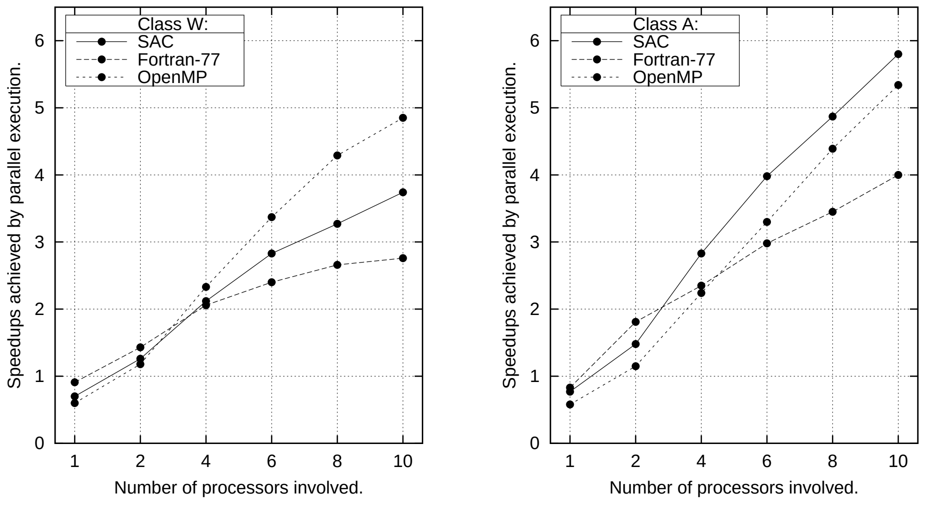


**Figure 13. Speedups relative to sequential Fortran-77 performance.**

memory management, on which SAC heavily relies. Since the absolute overhead incurred by memory management operations is invariant against grid sizes involved, it is negligible for large grids but shows a growing performance impact with decreasing grid size. As a consequence, the sequential kernel of NAS-MG, where operations are performed on very small grids, is considerably more expensive with dynamic memory management in SAC than it is with a static memory layout in a low-level Fortran-77 implementation or an almost static memory layout, as employed by the C/OpenMP code investigated. Fortunately, the absolute performance impact of this effect decreases with growing initial grid size. This explains why the scalability of the SAC code benefits significantly more from switching from size class W to size class A as the other implementations do.

## 6 Related Work

There are various approaches to raise the level of abstraction in array processing from that provided by conventional scalar languages. Fortran-90/95/HPF [1, 21] extend traditional Fortran-77 by a fixed set of built-in primitives which are applicable to entire arrays of any dimension and size. The *triple notation* allows to restrict such operations to subarrays and grids similar to WITH-loop generators. Still, arrays are manipulated via side-effecting operations and memory management is completely explicit. The language also provides no means to build generally applicable abstractions on top of the limited collection of array primitives.

Nevertheless, a considerable price in terms of runtime performance has to be paid for the increased level of abstraction. Investigations involving the NAS benchmark MG showed an HPF implementation to be outperformed by the reference Fortran-77+MPI solution by a factor of nearly three on a single processor and by a factor of 8 with both employing 32 processors [11, 12].

ZPL [7] offers a more elegant imperative solution based on *regions*. Regions are possibly dynamically defined sets of array indices, to which any scalar operation can be mapped to manipulate exactly the elements of the region. More sophisticated mappings, e.g. linear projections or permutations, can be realized through a set of built-in *prepositions*. Entire procedures can be applied to arrays of different dimension, but problems arise where the desired functionality depends on structural properties of arguments. Dimension-invariant abstractions are not possible.

Investigations on the NAS benchmark MG on a similar Sun Enterprise multiprocessor as was used in our experiments showed a maximum speedup of 5 using 14 processors, although the focus was on size classes B and C [8]. They represent significantly larger problems and, hence, should yield better parallel performance characteristics than size classes W and A used in our experiments. The investigations reported in [8] also cover SAC, showing it to be slightly inferior to ZPL both in sequential base performance as well as scalability. However, these results were obtained using previous versions of both the SAC compiler as well as the benchmark implementation.

In the field of functional programming languages, Sisal [6] used to be the most prominent array language. It offers high-level array handling free of side-effects based on implicit memory management. Compound array operations are defined by means of *for-loops*, Sisal-specific array comprehensions. However, the original design [23] supports only vectors; higher-dimensional arrays must be represented as nested vectors of equal length. It neither provides built-in high-level aggregate operations as, for instance, Fortran-90/95 nor means to define such general abstractions. More recent versions, e.g. Sisal 2.0 [5, 24] or Sisal-90 [10], promise improvements, but none of them have been implemented.

Previous investigations on the NAS benchmark MG have shown that SAC clearly outperforms Sisal in sequential execution [28]. However, these investigations differ from the results presented here in several aspects. The SAC implementation of NAS MG is replaced by new, significantly more generic code, the SAC compiler has considerably been improved since then, and, last but not least, multiprocessor performance using the implicit parallelization feature is analysed in addition to serial performance characteristics.

SA-C or SASSY [19] combines elements of Sisal's array support with a C-like syntax otherwise, the latter very much for the same reasons as SAC. SA-C provides specific support for image processing and explicitly targets reconfigurable computing systems based on FPGAs. The level of abstraction in array processing is similar to the later versions of Sisal. In particular, arrays always have a fixed dimension; generally applicable abstractions similar to those used for implementing NAS-MG in SAC are not supported. Due to the more specific target environment, no performance figures for implementations of the NAS benchmark suite are available for SA-C.

## 7 Conclusions and future work

This paper investigates the suitability of the functional array processing language SAC for implementing a nontrivial numerical problem. The NAS benchmark MG, which realizes multigrid relaxation with periodic boundary conditions, is chosen as a case study. A generic, high-level SAC implementation is presented, which in major parts almost literally follows the mathematical benchmark specification. Nevertheless, the SAC compiler succeeds in generating machine code which is outperformed by the low-level Fortran-77 reference implementation by only 23% and itself outperforms a C implementation which is directly derived from the reference implementation to a similar degree (size class A).

Using the implicit parallelization facility, SAC achieves speedups of up to 7.6 with 10 processors of a shared memory multiprocessor without any additional programming effort. It outperforms both automatically parallelized Fortran-77 code due to better scalability as well as C-based OpenMP code due to better sequential base performance. This is particularly remarkable as the SAC implementation is highly generic, solves the underlying problem at an almost mathematical level of abstraction, and reduces the code size compared with the two low-level solutions under consideration by more than an order of magnitude.

One area of future work are additional performance investigations. This includes larger problem sizes like size classes B and C of the NAS specification but also larger multiprocessor systems to determine scalability limits which have not yet been reached even for size class W. Furthermore, a direct comparison with the MPI-based parallel reference implementation of NAS-MG would be interesting.

Other areas of future work are improvements both in the benchmark as well as in the compiler implementation. A direct implementation of relaxation with periodic boundary conditions that makes artificial boundary elements obsolete is most desirable. On the one hand, it saves the overhead associated with updating these additional elements. On the other hand, it allows for a benchmark implementation that is even closer to the mathematical specification as the existing one. With respect to the compiler implementation, it would be interesting to investigate optimizations which implicitly realize the tricky stencil optimization exploited by both low-level solutions.

## References

[1] J. Adams, W. Brainerd, J. Martin, B. Smith, and J. Wagener. *Fortran-95 Handbook — Complete ANSI/ISO Reference*. Scientific and Engineering Computation. MIT Press, Cambridge, Massachusetts, USA, 1997.

[2] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, T. Schreiber, R. Simon, V. Venkatakrishnam, and S. Weeratunga. The NAS Parallel Benchmarks. *International Journal of Supercomputer Applications*, 5(3):63–73, 1991.

[3] D. Bailey, E. Barszcz, J. Barton, et al. The NAS Parallel Benchmarks. RNR 94-007, NASA Ames Research Center, Moffet Field, California, USA, 1994.

[4] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. NAS 95-020, NASA Ames Research Center, Moffet Field, California, USA, 1995.

[5] A. Böhm, D. Cann, R. Oldehoeft, and J. Feo. SISAL Reference Manual Language Version 2.0. CS 91-118, Colorado State University, Fort Collins, Colorado, USA, 1991.

[6] D. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, 1992.

[7] B. Chamberlain, S.-E. Choi, C. Lewis, L. Snyder, W. Weathersby, and C. Lin. The Case for High-Level Parallel Programming in ZPL. *IEEE Computational Science and Engineering*, 5(3), 1998.

[8] B. Chamberlain, S. Deitz, and L. Snyder. A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures. In *Proceedings of the ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC'00), Dallas, Texas, USA*. ACM Press and IEEE Computer Society Press, 2000.

[9] L. Dagum and R. Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Transactions on Computational Science and Engineering*, 5(1), 1998.

[10] J. Feo, P. Miller, S.K.Skedzielewski, S. Denton, and C. Solomon. Sisal 90. In A. Böhm and J. Feo, editors, *Proceedings of the Conference on High Performance Functional Computing (HPFC'95), Denver, Colorado, USA*, pages 35–47. Lawrence Livermore National Laboratory, Livermore, California, USA, 1995.

[11] M. Frumkin, H. Jin, and J. Yan. Implementation of NAS Parallel Benchmarks in High Performance Fortran. NAS Technical Report NAS-98-009, NASA Ames Research Center, Moffet Field, California, USA, 1998.

[12] M. Frumkin, H. Jin, and J. Yan. Implementation of NAS Parallel Benchmarks in High Performance Fortran. In *Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing, (IPPS/SPDP'99), San Juan, Puerto Rico*, 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, Berlin, Germany, 1999.

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

[15] C. Grelck and S.-B. Scholz. Accelerating APL Programs with SAC. In O. Lefevre, editor, *Proceedings of the International Conference on Array Processing Languages (APL'99), Scranton, Pennsylvania, USA*, volume 29 of *APL Quote Quad*, pages 50–57. ACM Press, 1999.

[16] C. Grelck and S.-B. Scholz. HPF vs. SAC — A Case Study. In A. Bode, T. Ludwig, W. Karl, and R. Wismüller, editors, *Proceedings of the 6th European Conference on Parallel Processing (Euro-Par'00), Munich, Germany*, volume 1900 of *Lecture Notes in Computer Science*, pages 620–624. Springer-Verlag, Berlin, Germany, 2000.

[17] W. Hackbusch. *Multigrid Methods and Applications*. Springer-Verlag, Berlin, Germany, 1985.

[18] W. Hackbusch and U. Trottenberg. *Multigrid Methods*, volume 960 of *Lecture Notes in Mathematics*. Springer-Verlag, Berlin, Germany, 1982.

[19] J. Hammes, B. Draper, and A. Böhm. Sassy: A Language and Optimizing Compiler for Image Processing on Reconfigurable Computing Systems. In H. Christensen, editor, *Proceedings of the International Conference on Vision Systems (ICVS'99), Las Palmas de Gran Canaria, Spain*, volume 1542 of *Lecture Notes in Computer Science*, pages 83–97. Springer-Verlag, Berlin, Germany, 1999.

[20] K. Iverson. *A Programming Language*. John Wiley, New York City, New York, USA, 1962.

[21] C. Koebel, D. Loveman, R. Schreiber, G. Steele, and M. Zosel. *The High Performance Fortran Handbook*. MIT Press, Cambridge, Massachusetts, USA, 1994.

[22] K. Kusano, S. Satoh, and M. Sato. Performance Evaluation of the Omni OpenMP Compiler. In *Proceedings of the International Workshop on OpenMP — Experiences and Implementations (WOMPEI'00), Yoyogi, Tokyo, Japan*, 2000.

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

[24] R. Oldehoeft. Implementing Arrays in SISAL 2.0. In *Proceedings of the 2nd SISAL Users Conference, San Diego, California, USA*, pages 209–222. Lawrence Livermore National Laboratory, 1992.

[25] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proceedings of the 1st European Workshop on OpenMP (EWOMP'99), Lund, Sweden*, pages 32–39, 1999.

[26] S.-B. Scholz. **S**ingle **A**ssignment **C** – Functional Programming Using Imperative Style. In J. Glauert, editor, *Proceedings of the 6th International Workshop on Implementation of Functional Languages (IFL'94), Norwich, England*, pages 21.1–21.13. University of East Anglia, Norwich, England, 1994.

[27] 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.

[28] S.-B. Scholz. A Case Study: Effects of WITH-Loop Folding on the NAS Benchmark MG in 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 216–228. Springer-Verlag, Berlin, Germany, 1999.