HPF vs. SAC — a Case Study

Clemens Grelck and Sven-Bodo Scholz

University of Kiel
Department of Computer Science and Applied Mathematics
e-mail: {cg,sbs}@informatik.uni-kiel.de

Abstract. This paper compares the functional programming language SAC to HPF with respect to specificational elegance and runtime performance. A well-known benchmark, red-black successive over-relaxation, serves as a case study. After presenting the HPF reference implementation alternative SAC implementations are discussed. Eventually, performance figures show the ability to compile highly generic SAC specifications into machine code that outperforms the HPF implementation on a shared memory multiprocessor.

1 Introduction

Programming language design basically is about finding the best possible tradeoff between support for high-level program specifications and runtime efficiency. In the context of array processing, data parallel languages are well-suited to meet this goal. Replacing loop nestings by language constructs that operate on entire arrays rather than on single elements, not only improves program specifications; it also creates new optimization opportunities for compilers [3, 4, 1, 8, 7].

FORTRAN-90/HPF introduce a large set of *intrinsics*, built-in operations that manipulate entire arrays in a homogeneous way and that are applicable to arrays of any dimensionality and size. While this allows for concise specifications of many algorithms, code becomes less generic if operations have to be applied to subsets of array elements only. Although regularly structured cases are addressed by the *triple notation*, one step back to loops and scalar specifications is often inevitable. In either case, the resulting code must be specialized to the shapes of argument arrays. Intrinsics are also limited to serve as primary building blocks of programs; FORTRAN-90/HPF provide no means to build abstractions upon intrinsics without loss of generality, i.e. applicability to arrays of any shape.

SAC is a functional C-variant with extended support for arrays [9]. It allows for high-level array processing similar to APL. The basic language construct for specifying array operations is the so-called WITH-*loop*. WITH-loops define `map`- or `fold`-like operations in a way that is invariant to the dimensionalities of argument arrays. As a consequence, almost all operations, typically found as built-in functions in other array languages, can be defined through WITH-loops in SAC without any loss of generality [6]. This concept allows for both: comprehensive array support through easily maintainable libraries and far-reaching customization opportunities for programmers.

In Section 2 we investigate the specificational benefits of SAC in terms of generic high-level programming compared to HPF. In Section 3, we find out how much of a performance penalty has actually to be paid for the increased level of abstraction. Since the SAC-compiler allows to implicitly generate code for shared memory multiprocessors [5], we focus on this architecture. Eventually, Section 4 concludes.

2 A Case Study: the PDE1-Benchmark

As reference implementation for the case study, we chose the PDE1-benchmark as it is supplied by the distribution of the ADAPTOR HPF compiler [2]. PDE1 is a red-black SOR for approximating three-dimensional Poisson equations. The core of the algorithm is a stencil operation on a three-dimensional array u: for each inner element $u_{i,j,k}$, the values of the 6 direct neighbor elements are summed up, added to a fixed number $h^2 f_{i,j,k}$, and subsequently multiplied with a constant factor. Assuming `NX`, `NY`, and `NZ` to denote the extents of the three-dimensional arrays `U`, `U1`, and `F`, this operation in the reference implementation is specified as:

```
  U1(2:NX-1,2:NY-1,2:NZ-1) =                                 &
&             FACTOR*(HSQ*F(2:NX-1,2:NY-1,2:NZ-1)+           &
&    U(1:NX-2,2:NY-1,2:NZ-1)+U(3:NX,2:NY-1,2:NZ-1)+          &
&    U(2:NX-1,1:NY-2,2:NZ-1)+U(2:NX-1,3:NY,2:NZ-1)+          &
&    U(2:NX-1,2:NY-1,1:NZ-2)+U(2:NX-1,2:NY-1,3:NZ))
```

However, this operation has to be applied to two disjoint sets of elements (the *red* elements and the *black* elements) in two successive steps. This is realized by creating a three-dimensional array of booleans `RED` and embedding the array assignment shown above into a `WHERE` construct.

The given HPF solution can be carried over to SAC almost straightforwardly. Rather than using the triple notation of HPF, in SAC, the computation of the inner elements is specified for a single element at index position `iv`, which by means of a WITH-loop is mapped to all inner elements of an array `u`:

```
u1 = with (. < iv < .) {
        st_sum = u[iv+[1,0,0]] + u[iv-[1,0,0]] + u[iv+[0,1,0]]
               + u[iv-[0,1,0]] + u[iv+[0,0,1]] + u[iv-[0,0,1]];
     } modarray (u, iv, factor * (hsq * f[iv] + st_sum));
```

Note here, that the usage of `<` instead of `<=` on both sides of the generator part restricts the elements to be computed to the inner elements of the array `u`.

The disadvantage of this solution is that it is tailor-made for the given stencil. In the same way the access triples in the HPF-solution have to be adjusted whenever the stencil changes, the offset vectors have to be adjusted in the SAC solution. These adjustments are very error-prone; in particular, if the size of the stencil increases or the dimensionality of the problem has to be changed. To alleviate these problems, we abstract from the problem specific part by introducing an array of weights `W`. In this particular example, `W` is an array of shape `[3,3,3]` with all elements being `0` but the six direct neighbor elements of the center element, which are set to `1`. With such an array `W`, relaxation can be defined as:

```
u1 = with (. < iv < .) {
       block = tile( shape(W), iv-1, u);
     } modarray( u, iv, factor * (hsq * f[iv] + sum( W * block)));
```

In this specification, for each inner element of `u1` a sub-array `block` is taken from `u` which holds all the neighbor elements of `u[iv]`. This is done by applying the library function `tile(` *`shape`*`,` *`offset`*`,` *`array`*`)` which creates an array of shape *shape* whose elements are taken from *array* starting at position *offset*. The computation of the weighted sum of neighbor elements thus turns into `sum( W * block)`, where `(` *`array`* `*` *`array`* `)` refers to an elementwise multiplication of arrays, and `sum(` *`array`*`)` sums up all elements of *array*.

Abstracting from the problem specific stencil data has another advantage: the resulting program does not only support arbitrary stencils but can also be applied to arrays and stencils of other dimensionalities without changes. Note here, that the usage of `shape(W)` rather than `[3,3,3]` as first argument for `tile` is essential for achieving this.

Although the error-prone indexing operations have been eliminated by the introduction of `W`, the specification still consists of a problem specific WITH-loop which contains an elementwise specification of the relaxation step. It should be noted here, that the elementwise specification can be "lifted" into a nesting of APL-like operations on entire arrays without causing any performance penalty (cf. [?]).

After defining relaxation on the entire array, the operation has to be restricted to subsets of the array elements, i.e. to the sets of red and black elements. In the same way as in the HPF program, an array of booleans can be defined which masks the elements of the red set.

For avoiding computational redundancy, the restriction to red/black elements in the HPF solution is realized by integrating it into relaxation algorithm itself. In SAC, we want to keep these specifications separated in order to improve program modularity as well as its potential for code re-use. Therefore, a shape-invariant general purpose function `CombineMasked( mask, a, b)` is defined, which according to a mask of booleans combines two arrays into a new one:

```
inline double[] CombineMasked( bool[] mask, double[] a, double[] b)
{
  c = with(. <= iv <= .)
      genarray( shape(a), (mask[iv]? a[iv]: b[iv]));
  return( c);
}
```

Provided that `mask`, `a`, and `b` are identically shaped, a new array `c` of the same shape is created, whose elements are copied from those of the array `a` if the mask is true, and from `b` otherwise. Using this function, red black relaxation can be defined as:

```
  u = CombineMasked( red, relax(u, f, hsq), u);
  u = CombineMasked( !red, relax(u, f, hsq), u);
```

Note here, that the black set is referred to by `!red`, i.e., by using the elementwise extension of the negation operator (`!`).

3 Performance Comparison

This section presents the essence of thorough investigations on the performance of the HPF- and various alternative SAC-implementations of PDE1 on a 12-processor SUN Ultra Enterprise 4000. The ADAPTOR HPF-compiler v7.0 [2], Sun f77 v5.0, and PVM 3.4.2 for shared memory were used to evaluate the HPF code, the SAC compiler v0.9 and Sun cc v5.0 to compile the SAC code.

single node performance		
runtime	HPF	SAC
64^3	283ms	84ms
256^3	22.2s	6.6s
memory	HPF	SAC
64^3	10MB	8MB
256^3	450MB	260MB

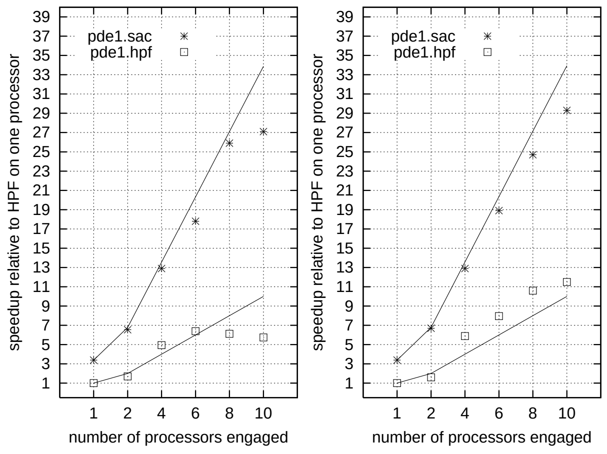

Fig. 1. Runtime performance of SAC- and HPF-implementations of the PDE1 benchmark, problem sizes 64^3 (center) and 256^3 (right).

One interesting result is that with respect to the accuracy of the timing facility all different SAC-specifications — among them those presented in Section 2 — achieve the same runtimes. Having a look into the compiled code reveals that the SAC-compiler manages to transform all of them into almost identical intermediate representations. This is mostly due to a SAC-specific optimization technique called WITH-LOOP-FOLDING [10] that aggressively eliminates intermediate arrays. Fig. 1 shows performance results for the problem sizes 64^3 and 256^3. Upon sequential execution, SAC outperforms HPF by a factor of 3.4 for both problem sizes; SAC also needs much less memory: 260MB instead of 450MB in the 256^3 case. This decrease in memory consumption can also be attributed to WITH-LOOP-FOLDING.

Multiprocessor runtimes of the HPF- and SAC-code are shown as speedups relative to HPF single node runtimes. For 64^3 elements, HPF scales well up to 6 processors; any additional processor leads to absolute performance degradation. In contrast, the SAC runtimes scale linearly up to 8 processors and even achieve an additional speedup with all 10 processors. The HPF performance scales much better for the problem size 256^3. So, the usage of PVM as low-level communication layer is no principle hindrance to achieve good performance on a shared memory architecture. Nevertheless, even with 10 processors SAC outperforms HPF by a factor of 2.5.

4 Conclusion

The major design goal of SAC is to combine highly generic specifications of array operations with compilation techniques for generating efficiently executable code. By means of a case study, this paper investigates different opportunities for the specification of the PDE1 benchmark in SAC and compares them to the HPF reference implementation in terms of specificational elegance and reusability. Despite their increasingly higher levels of abstraction the various SAC implementations clearly outperform the given HPF program on a shared memory multiprocessor. This shows that high-level generic program specifications and good runtime performance not necessarily exclude each other.

References

1. G.E. Blelloch, S.Chatterjee, J.C. Hardwick, J. Sipelstein, and M.Zagha. Implementation of a Portable Nested Data-Parallel Language. In *Proceedings 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, California*, pages 102–111, 1993.
2. T. Brandes and F. Zimmermann. ADAPTOR - A Transformation Tool for HPF Programs. In *Programming Environments for Massively Parallel Distributed Systems*, pages 91–96. Birkhäuser Verlag, 1994.
3. D.C. Cann. Compilation Techniques for High Performance Applicative Computation. Technical Report CS-89-108, Lawrence Livermore National Laboratory, LLNL, Livermore, California, 1989.
4. D.C. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, 35(8):81–89, 1992.
5. C. Grelck. Shared Memory Multiprocessor Support for SAC. In K. Hammond, T. Davie, and C. Clack, editors, *Proc. of Implementing Functional Languages (IFL '98), London, Selected Papers*, volume 1595 of *LNCS*, pages 38–54. Springer, 1999.
6. C. Grelck and S.-B. Scholz. Accelerating APL Programs with SAC. In O. Lefevre, editor, *Proceedings of the Array Processing Language Conference (APL'99), Scranton, Pa.*, volume 29(1) of *APL Quote Quad*, pages 50–57. ACM Press, 1999.
7. E.C. Lewis, C. Lin, and L. Snyder. The Implementation and Evaluation of Fusion and Contraction in Array Languages. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation.* ACM, 1998.
8. G. Roth and K. Kennedy. Dependence Analysis of Fortran90 Array Syntax. In *Proc. PDPTA'96*, 1996.
9. S.-B. Scholz. **S***ingle* **A***ssignment* **C** – *Entwurf und Implementierung einer funktionalen C-Variante mit spezieller Unterstützung shape-invarianter Array-Operationen.* PhD thesis, University of Kiel, 1996.
10. S.-B. Scholz. With-loop-folding in SAC–Condensing Consecutive Array Operations. In C. Clack, K.Hammond, and T. Davie, editors, *Implementation of Functional Languages, 9th International Workshop, IFL'97, St. Andrews, Scotland, UK, September 1997, Selected Papers*, volume 1467 of *LNCS*, pages 72–92. Springer, 1998.