# Towards an Efficient Functional Implementation of the NAS Benchmark FT

Clemens Grelck[1] and Sven-Bodo Scholz[2]

[1] University of Lübeck, Germany
Institute of Software Technology and Programming Languages
grelck@isp.uni-luebeck.de
[2] University of Kiel, Germany
Institute of Computer Science and Applied Mathematics
sbs@informatik.uni-kiel.de

**Abstract.** This paper compares a high-level implementation of the NAS benchmark FT in the functional array language SaC with traditional solutions based on Fortran-77 and C. The impact of abstraction on expressiveness, readability, and maintainability of code as well as on clarity of underlying mathematical concepts is discussed. The associated impact on runtime performance is quantified both in a uniprocessor environment as well as in a multiprocessor environment based on automatic parallelization and on OpenMP.

## 1 Introduction

Low-level sequential base languages, e.g. Fortran-77 or C, and message passing libraries, mostly Mpi, form the prevailing tools for generating parallel applications, in particular for numerical problems. This choice offers almost literal control over data layout and program execution, including communication and synchronization. Expertised programmers are enabled to adapt their code to hardware characteristics of target machines, e.g. properties of memory hierarchies, and to enhance the runtime performance to whatever a machine is able to deliver.

During the process of performance tuning, numerical code inevitably mutates from a (maybe) human-readable representation of an abstract algorithm to one that almost certainly is suitable for machines only. Ideas and concepts of underlying mathematical algorithms are completely disguised. Even minor changes to underlying algorithms may require a major re-design of the implementation. Moreover, particular demand is made on the qualification of programmers as they have to be experts in computer architecture and programming technique in addition to their specific application domains. As a consequence, development and maintenance of parallel code is prohibitively expensive.

As an alternative approach, functional languages encourage a declarative style of programming that abstracts from many details of program execution. For example, memory management for aggregate data structures like arrays is completely up to compilers and runtime systems. Even arrays are stateless and may be passed to and from functions following a call-by-value semantics. Focusing on algorithmic rather than on organizational aspects, functional languages significantly reduce the gap between a mathematical idea and an executable specification; their side-effect free semantics facilitates parallelization [1].

Unfortunately, in numerical computing functional languages have shown performance characteristics inferior to well-tuned (serial) imperative codes to an extent which renders parallelization unreasonable [2]. This observation has inspired the design of the functional array language SaC [3]. SaC (for Single Assignment C) aims at combining high-level program specifications characteristic for functional languages with efficient support for array processing in the style of Apl including automatic parallelization (for shared memory systems at the time being) [4,5]. Efficiency concerns are addressed by incorporating both well-known and language-specific optimization techniques into the SaC compiler, where their applicability significantly benefits from the side-effect free, functional semantics of the language[1].

This paper investigates the trade-off between programming productivity and runtime performance by means of a single though representative benchmark: the application kernel FT from the NAS benchmark suite [6]. Investigations on this benchmark involving the functional languages Id [7] and Haskell [8] have contributed to a pessimistic assessment of the suitability of functional languages for numerical computing in general [2]. We show a very concise, almost mathematical SaC specification of NAS-FT, which gets as close as within a factor of 2.8 to the hand-tuned, low-level Fortran-77 reference implementation and outperforms that version by implicitly using four processors of a shared memory multiprocessor system.

## 2 Implementing the NAS Benchmark FT

The NAS benchmark FT implements a solver for a class of partial differential equations by means of repeated 3-dimensional forward and inverse complex fast-Fourier transforms. They are implemented by consecutive collections of 1-dimensional FFTs on vectors along the three dimensions, i.e., an array of shape `[X,Y,Z]` is consecutively interpreted as a `ZY` matrix of vectors of length `X`, as a `ZX` matrix of vectors of length `Y`, and as a `XY` matrix of vectors of length `Z`.

The outline of this algorithm can be carried over into a SaC specification straightforwardly, as shown in Fig. 1. The function `FFT` on 3-dimensional complex arrays (`complex[.,.,.]`) consecutively transposes the argument array `a` three times. After each transposition, the function `Slice` extracts all subvectors along the innermost axis and individually applies 1-dimensional FFTs to them. The additional parameter `rofu` provides a pre-computed vector of complex roots of unity, which is used for 1-dimensional FFTs. The 3-line definition of `Slice` is omitted here for space reasons and because it requires more knowledge of SaC.

The overloaded function `FFT` on vectors of complex numbers (`complex[.]`) almost literally implements the Danielson-Lanczos algorithm [9]. It is based on

[1] More information on SaC is available at `http://www.sac-home.org/` .

```
complex[.,.,.] FFT( complex[.,.,.] a, complex[.] rofu)
{
  a_t = transpose( [2,1,0], a);
  b   = Slice( FFT, a_t, rofu);

  b_t = transpose( [0,2,1], b);
  c   = Slice( FFT, b_t, rofu);

  c_t = transpose( [1,2,0], c);
  d   = Slice( FFT, c_t, rofu);

  return( d);
}

complex[.] FFT(complex[.] v, complex[.] rofu)
{
  even      = condense(2, v);
  odd       = condense(2, rotate( [-1], v));
  rofu_even = condense(2, rofu);

  fft_even  = FFT( even, rofu_even);
  fft_odd   = FFT( odd,  rofu_even);

  left      = fft_even + fft_odd * rofu;
  right     = fft_even - fft_odd * rofu;

  return( left ++ right);
}

complex[2] FFT(complex[2] v, complex[1] rofu)
{
  return( [ v[0] + v[1], v[0] - v[1] ]);
}
```

**Fig. 1.** SaC implementation of NAS-FT.

the recursive decomposition of the argument vector v into elements at even and at odd index positions. The vector even can be created by means of the library function condense(n,v), which selects every n-th element of v. The vector odd is generated in the same way after first rotating v by one index position to the left. FFT is then recursively applied to even and to odd elements, and the results are combined by a sequence of element-wise arithmetic operations on vectors of complex numbers and a final vector concatenation (++). A direct implementation of FFT on 2-element vectors (complex[2]) terminates the recursion. Note that unlike Fortran neither the data type complex nor any of the operations used to define FFT are built-in in SaC; they are all are imported from the standard library, where they are defined in SaC itself.

In order to help assessing the differences in programming style and abstraction, Fig. 2 shows excerpts from about 150 lines of corresponding Fortran-77 code. Three slightly different functions, i.e. cffts1, cffts2, and cffts3, intertwine the three transposition operations with a block-wise realization of a 1-dimensional FFT. The iteration is blocked along the middle dimension to improve cache performance. Extents of arrays are specified indirectly to allow reuse of the same set of buffers for all orientations of the problem. Function fftz2 is part of the 1-dimensional FFT. It must be noted that this excerpt represents high quality code, which is well organized and well structured. It was written by

```
subroutine  cffts1 ( is,d,x,xout,y)

include 'global.h'
integer is, d(3), logd(3)
double complex x(d(1),d(2),d(3))
double complex xout(d(1),d(2),d(3))
double complex y(fftblockpad, d(1), 2)
integer i, j, k, jj

do i = 1, 3
 logd(i) = ilog2(d(i))
end do

do k = 1, d(3)
 do jj = 0, d(2)-fftblock, fftblock
  do j = 1, fftblock
   do i = 1, d(1)
    y(j,i,1) = x(i,j+jj,k)
   enddo
  enddo

  call cfftz (is, logd(1),
              d(1), y, y(1,1,2))

  do j = 1, fftblock
   do i = 1, d(1)
    xout(i,j+jj,k) = y(j,i,1)
   enddo
  enddo
 enddo
enddo

return
end
```

```
subroutine fftz2 ( is,l,m,n,ny,ny1,u,x,y)

integer is,k,l,m,n,ny,ny1,n1,li,lj
integer lk,ku,i,j,i11,i12,i21,i22
double complex u,x,y,u1,x11,x21
dimension u(n), x(ny1,n), y(ny1,n)

n1 = n / 2
lk = 2 ** (l - 1)
li = 2 ** (m - 1)
lj = 2 * lk
ku = li + 1

do i = 0, li - 1
 i11 = i * lk + 1
 i12 = i11 + n1
 i21 = i * lj + 1
 i22 = i21 + lk
 if (is .ge. 1) then
  u1 = u(ku+i)
 else
  u1 = dconjg (u(ku+i))
 endif
 do k = 0, lk - 1
  do j = 1, ny
   x11 = x(j,i11+k)
   x21 = x(j,i12+k)
   y(j,i21+k) = x11 + x21
   y(j,i22+k) = u1 * (x11 - x21)
  enddo
 enddo
enddo

return
end
```

**Fig. 2.** Excerpts from the FORTRAN-77 implementation of NAS-FT.

expert programmers in the field and has undergone several revisions. Everyday legacy FORTRAN-77 code is likely to be less "intuitive".

## 3 Experimental Evaluation

This section compares the runtime performance achieved by code compiled from the high-level functional SAC specification of NAS-FT, as outlined in the previous section, with that of two low-level solutions: the serial FORTRAN-77 reference implementation[2] and a C implementation derived from the reference code and extended by OPENMP directives by Real World Computing Partnership (RWCP)[3]. All experiments were made on a 12-processor SUN Ultra Enterprise 4000 shared memory multiprocessor using SUN Workshop compilers. Investigations covered size classes W and A; as the findings were almost identical, we focus on size class A in the following.

As shown in Fig. 3, SAC is outperformed by the FORTRAN-77 reference implementation by not more than a factor of 2.8 and by the corresponding C code by a factor of 2.4. To a large extent, this can be attributed to dynamic memory management overhead caused by the recursive decomposition of argument vectors when computing 1-dimensional FFTs. In contrast to SAC, both the FORTRAN-77 and the C implementation use a static memory layout.

[2] The source code is available at `http://www.nas.nasa.gov/Software/NPB/` .

[3] The source code is available at `http://phase.etl.go.jp/Omni/` .

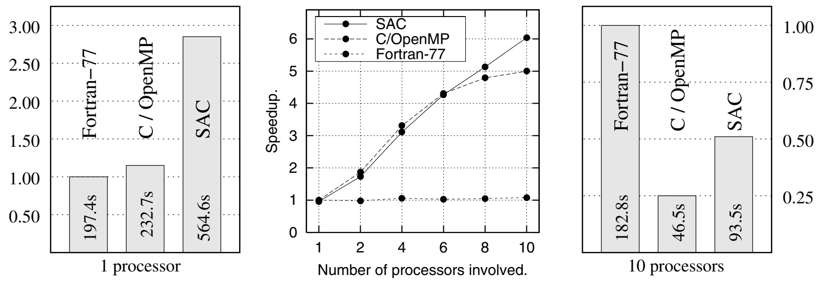

**Fig. 3.** Runtime performance of NAS-FT: sequential, scalability, ten processors.

Fig. 3 also reports on the scalability of parallelization, i.e. parallel execution times divided by each candidate's best serial runtime. Whereas hardly any performance gain can be observed for automatic parallelization of the FORTRAN-77 code by the SUN Workshop compiler, SAC achieves speedups of up to six. Hence, SAC equalizes FORTRAN-77 with four processors and outperforms it by a factor of about two when using ten processors. SAC even scales slightly better than OPENMP. This is remarkable as the parallelization of SAC code is completely implicit, whereas a total of 25 compiler directives guide parallelization in the case of OPENMP. However, it must also be mentioned that the C/OPENMP solution achieves the shortest absolute 10-processor runtimes due to its superior sequential performance.

## 4 Related Work and Conclusions

There are various approaches to raise the level of abstraction in array processing from that provided by conventional scalar languages. FORTRAN-90 and ZPL [10] treat arrays as conceptual entities rather than as loose collections of elements. Although they do not at all reach a level of abstraction similar to that of SAC, a considerable price in terms of runtime performance has to be paid [11]. SISAL [12] used to be the most prominent functional array language. However, apart from a side-effect free semantics and implicit memory management the original design provides no support for high-level array processing in the sense of SAC. More recent versions [13] promise improvements, but have not been implemented.

General-purpose functional languages offer a significantly more abstract programming environment. However, investigations involving HASKELL [8] and ID [7] based on the NAS benchmark FT revealed substantial deficiencies both in time and space consumption [2]. Our experiments showed that HASKELL implementations described in [2] are outperformed by the FORTRAN-77 reference implementation by more than two orders of magnitude for size class W. Experiments on size class A failed due to memory exhaustion.

The development of SAC aims at combining high-level functional array programming with competitive runtime performance. The paper evaluates this approach based on the NAS benchmark FT. It is shown how 3-dimensional FFTs can be assembled by about two dozen lines of SAC code as opposed to 150 lines of fine-tuned Fortran-77 code in the reference implementation. Moreover, the SaC solution clearly exhibits underlying mathematical ideas, whereas they are completely disguised by performance-related coding tricks in the case of Fortran. Nevertheless, the runtime of the SaC implementation is within a factor of 2.8 of the Fortran code. Furthermore, the SaC version without any modification outperforms its Fortran counterpart on a shared memory multiprocessor as soon as four or more processors are used. In contrast, additional effort and knowledge are required for the imperative solution to effectively utilize the SMP system. Annotation with 25 OpenMP directives succeeded in principle, but did not scale as good as the compiler-parallelized SaC code.

## References

1. Hammond, K., Michaelson, G. (eds.): Research Directions in Parallel Functional Programming. Springer-Verlag (1999)
2. Hammes, J., Sur, S., Böhm, W.: On the Effectiveness of Functional Language Features: NAS Benchmark FT. Journal of Functional Programming **7** (1997) 103–123
3. Scholz, S.B.: Single Assignment C — Efficient Support for High-Level Array Operations in a Functional Setting. Journal of Functional Programming, accepted for publication
4. Grelck, C.: Shared Memory Multiprocessor Support for SAC. In: Hammond, K., Davie, D., Clack, C. (eds.): Implementation of Functional Languages. Lecture Notes in Computer Science, Vol. 1595. Springer-Verlag (1999) 38–54
5. Grelck, C.: A Multithreaded Compiler Backend for High-Level Array Programming. In: Proc. 21st International Multi-Conference on Applied Informatics (AI'03), Part II: International Conference on Parallel and Distributed Computing and Networks (PDCN'03), Innsbruck, Austria, ACTA Press (2003) 478–484
6. Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. NAS 95-020, NASA Ames Res. Center (1995)
7. Nikhil, R.: The Parallel Programming Language ID and its Compilation for Parallel Machines. In: Proc. Workshop on Massive Parallelism: Hardware, Programming and Applications, Amalfi, Italy, Academic Press (1989)
8. Peyton Jones, S.: Haskell 98 Language and Libraries. Cambridge University Press (2003)
9. Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: Numerical Recipes in C. Cambridge University Press (1993)
10. Chamberlain, B., Choi, S.E., Lewis, C., Snyder, L., Weathersby, W., Lin, C.: The Case for High-Level Parallel Programming in ZPL. IEEE Computational Science and Engineering **5** (1998)
11. Frumkin, M., Jin, H., Yan, J.: Implementation of NAS Parallel Benchmarks in High Performance Fortran. In: Proc. 13th International Parallel Processing Symposium/ 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP'99), San Juan, Puerto Rico. (1999)
12. Cann, D.: Retire Fortran? A Debate Rekindled. Communications of the ACM **35** (1992) 81–89
13. Feo, J., Miller, P., S.K.Skedzielewski, Denton, S., Solomon, C.: Sisal 90. In: Proc. Conference on High Performance Functional Computing (HPFC'95), Denver, Colorado, USA. (1995) 35–47