Implementing a Numerical Solution of the KPI Equation Using Single Assignment C: Lessons and Experiences

Alex Shafarenko[1], Sven-Bodo Scholz[1], Stephan Herhut[1], Clemens Grelck[2], and Kai Trojahner[2]

[1] Department of Computer Science, University of Hertfordshire, AL10 9AB, U.K.
[2] Institute of Software Technology and Programming Languages, University of Lübeck, Germany

Abstract. We report our experiences of programming in the functional language SAC[1] a numerical method for the KPI (Kadomtsev-Petiviashvili I) equation. KPI describes the propagation of nonlinear waves in a dispersive medium. It is an integro-differential, nonlinear equation with third-order derivatives, and so it presents a noticeable challenge in numerical solution, as well as being an important model for a range of topics in computational physics. The latter include: long internal waves in a density-stratified ocean, ion-acoustic waves in a plasma, acoustic waves on a crystal lattice, and more. Thus our solution of KPI in SAC represents an experience of solving a "real" problem using a single-assignment language and as such provides an insight into the kind of challenges and benefits that arise in using the functional paradigm in computational applications. The paper describes the structure and functionality of the program, discusses the features of functional programming that make it useful for the task in hand, and touches upon performance issues.

1 Introduction

It is common knowledge that the uptake of the functional programming technology is impeded by the lack of convincing evidence of the functional paradigm efficacy and suitability of expression. There is a considerable interest in seeing so-called 'real-life' applications programmed in a functional language, especially where these implementations show acceptable run-time performance and design advantages of the functional programming method.

In functional programming, component algorithms, rather than whole problems, tend to be used as benchmarks. We ourselves evaluated the performance of the Fast Fourier Transform component in the past [2] and so did the authors of [3]; paper [4] uses the conjugate gradient method as a benchmark, and the authors of [5] study the intricacies of matrix multiplication.

There is a significant advantage in using a whole application rather than a component algorithm as a benchmark. Firstly, it provides a balance of design patterns that may reflect more adequately the mix of methods, access patterns

A. Butterfield, C. Grelck, and F. Huch (Eds.): IFL 2005, LNCS 4015, pp. 160–177, 2006.

and programming techniques characteristic of a real-life programming project. Secondly, one stands a better chance of discovering situations in which the quality of expression or indeed the quality of generated code is properly challenged, so that one may learn some important lessons.

This paper has precisely such intent. We have selected a problem in computational mathematics which is complex enough to be interesting, yet not too complex, so that we are able to present the results and explain the design decisions in a short conference paper. The rest of the paper is organised as follows. The next Section introduces the equation and the solution method, Section 3 discusses the SaC implementation issues, Section 4 presents the conclusions we have drawn from implementing the solution method in SaC, next Section briefly discusses our equivalent Fortran code, Section 6 presents the results of performance studies involving several platforms and commercial compilers as a basis for comparison, and finally there are some conclusions.

2 The Equation

For this study we chose a problem that one of the authors had been familiar with from the time some 20 years back when he was doing his PhD in computational physics at University of Novosibirsk[6]: a Kadomtsev-Petviashvili equation. The KPI (Kadomtsev Petviashvili I) has the following canonic form:

$$\frac{\partial}{\partial x}\left(\frac{\partial u}{\partial t} + 6u\frac{\partial u}{\partial x} + \frac{\partial^3 u}{\partial x^3}\right) - 3\frac{\partial^2 u}{\partial y^2} = 0\,.$$

For computational reasons, it is more convenient to use the equivalent form of KPI:

$$\frac{\partial u}{\partial t} + 6u\frac{\partial u}{\partial x} + \frac{\partial^3 u}{\partial x^3} - 3\int_{-\infty}^{x}\frac{\partial^2 u}{\partial y^2} = 0$$

which can be written as

$$\frac{\partial u}{\partial t} = N(u) + L(u)\,,$$

where

$$N(u) = -6u\frac{\partial u}{\partial x} + 3\int_{-\infty}^{x}\frac{\partial^2 u}{\partial y^2} \qquad L(u) = \frac{\partial^3 u}{\partial x^3}\,,$$

are the nonlinear, diffractive part and the dispersion term, respectively.

The KPI model is very general indeed. It describes any physical system in which waves propagate mostly in one direction, but suffer from diffraction, i.e. the divergence of a wave packet across the propagation direction; dispersion, i.e. the widening of the wave along the propagation direction due to different parts of it propagating at different velocities; and hydrodynamic non-linearity, i.e. the fact that the wave tends towards steeper and steeper shapes until it either breaks or the steepening is arrested by the dispersion effects.

To give an idea of where the KPI model may apply, we quote its original application to water surface waves[7], its use as a model of ion-acoustic waves in plasma[8] and the application to string theory in high-energy physics[9], but also such a down-to-earth area as computing wave-resistance of a ship that travels at high speed along a waterway [10].

The numerical method for this paper has been borrowed from [11], except the boundary conditions whose discretisation was not defined there, so we used one of our own, bearing in mind that its effect on performance is insignificant.

The spatial derivatives in L and N were discretised thus:

$$-\frac{\partial^3 u}{\partial x^3} \to -\frac{u_{i+2,j} - 2u_{i+1,j} + 2u_{i-1,j} - u_{i-2,j}}{2\Delta x^3},$$

$$\frac{\partial u}{\partial x} \to \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x}$$

and

$$\frac{\partial^2 u}{\partial y^2} \to \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{2\Delta y}.$$

The integration along x was discretised by Simpson method and modified to take account of the boundary conditions.

The resulting scheme can be summarized as follows:

$$u^{n+1/3} = u^n + \gamma_1 \Delta t N(u^n) + \alpha_1 \Delta t \frac{L(u^{n+1/3}) + L(u^n)}{2},$$

$$u^{n+2/3} = u^{n+1/3} + \gamma_2 \Delta t N(u^{n+1/3}) + \rho_1 \Delta t N(u^n) + \alpha_2 \Delta t \frac{L(u^{n+2/3}) + L(u^{n+1/3})}{2},$$

$$u^{n+1} = u^{n+2/3} + \gamma_3 \Delta t N(u^{n+2/3}) + \rho_2 \Delta t N(u^n) + \alpha_3 \Delta t \frac{L(u^{n+1}) + L(u^{n+2/3})}{2},$$

which accounts for the Crank-Nicholson representation of the diffractive term and the conventional Runge-Kutta time-integration to the third order of accuracy. Here all α,γ and ρ are scalar constants chosen to achieve the required approximation accuracy. The boundary conditions are periodic in y and absorbing $\frac{\partial^2 u}{\partial x^2} = 0$ in x at both ends of the interval. The upper indices $n+1/3$, $n+2/3$, and $n+1$ refer to the 3 substeps of time-integration that make up a full step.

3 Implementation

With the above equations as a starting point, we identify the following tasks. First of all, since the above scheme is implicit in x, and a 5-point stencil is used for L, a pentdiagonal solver is required for all three substeps. The solver (which is a re-write of [12]) is a particular case of the LU-decomposition solver, taking advantage of the fact that the matrix only has five nonzero diagonals to reduce the solution complexity from $O(n^3)$ down to $O(n)$ by recurrent substitution. While this algorithm is recurrent in x, it is fully data-parallel in y. Hence we

decided to produce a one-dimensional implementation of the solver (the recurrences in question) which will take as many additional axes as required by the environment. The number of additional axes in our case would be one, since we are focusing on the two-dimensional KPI; however the equation itself is defined for three dimensions as well, hence the aforementioned additional flexibility is quite important for developing a future-proof program. Next, the pentdiagonal solver has, naturally, an elimination and a back-substitution phase, of which the former can be pre-computed (save for the right-hand side), thanks to the linearity of the scheme in the third derivative. Hence we need two functions, one for the eliminator and one for the rest of the solver. The eliminator is displayed in Fig. 1. Notice that the code is rank-monomorphic, in that it expects a fixed rank of its arguments a, b, c, d, e, which are the contents of five nonzero diagonals of the equation matrix, and in that it produces four fixed-rank arrays. Contrast that with the main solver, presented in Fig. 2. Here the result has undeclared rank, which can be 1,2, or more, which is decided on the basis of the shape of the similarly undeclared right-hand side f. The function *pent* will ensure that the shape of the result agrees with the shape of the argument; in any case, only the first dimension of both f and the result is explicitly referred to in the code. The very significant advantage of that has been that we could fully test this function on short one-dimensional data and then use it in the program for 2d arguments without any uncertainty as to its correctness in that case. Such rank invariance is not available with conventional array programming using, e.g., FORTRAN-90.

Similarly we designed a little function for Simpson integration, Fig. 3, which is rank-invariant, so it could be fully tested in a single dimension and then applied in two dimensions as the scheme demands.

Figure 14 displays the main program. It defines several constants and creates the first copy of u, which is the field array, by setting its shape and filling it in with the known soliton distribution (for which the SAC code is not shown). Next, it prepares the 5 diagonals a-e for the solver taking into account the boundary conditions. This results in a code pattern whereby first an array is initialised with the regular value, and then the boundaries are set by specific definitions. Notice that the five arrays are in fact two-dimensional, which has nothing to do with the two dimensions of the KPI equation, but merely reflects the fact that the numerical scheme has three substeps, so it is convenient to initialize the diagonals for all three substeps at once (by grouping them into a dimension) and then use the correct vector at each substep. This is achieved by using a 3-element vector *eps* in defining the default value for each of the diagonals. Finally the eliminator *prepent* is run for all substeps simultaneously, using the second dimension of the diagonal arrays.

The actual time stepping is performed by a for-loop at the end of the main function. During the step the array u is redefined three times, each time with the corresponding scheme formula. With little difficulty, one can see the original mathematics by looking at the program. The main discrepancy is the choice of indices for the constants α,γ and ρ which have to start from 0 since that is the C

```
inline
  double[.],                /* p */
  double[.],                /* q */
  double[.],                /* bet */
  double[.]                 /* den */
prepent( double[.] a,
         double[.] b,
         double[.] c,
         double[.] d,
         double[.] e)
{
  n = shape( a)[0];
  buf = genarray( [n], undef);
  p = buf; q = buf; bet = buf; den = buf;

  bet[0] = 1.0 / c[0];
  p[0] = -d[0] * bet[0];
  q[0] = -e[0] * bet[0];

  bet[1] = -1.0 / ( c[1] + b[1] * p[0]);
  p[1] = ( d[1] + b[1] * q[0]) * bet[1];
  q[1] = e[1] * bet[1];
  den[1] = b[1];

  for ( i=2; i<n; i++) {
    bet[i] = b[i] + a[i] * p[i-2];
    den[i] = -1.0 / ( c[i] + a[i] * q[i-2] +
                      bet[i] * p[i-1]);
    p[i] = ( d[i] + bet[i] * q[i-1]) * den[i];
    q[i] = e[i] * den[i];
  }

  return( p, q, bet, den);
}
```

Fig. 1. SAC code of the eliminator of the pentdiagonal solver

convention, while they start from 1 in the algorithm. Other than that, we have managed to represent the numerical scheme well near ditto.

4 Lessons and Conclusions

First of all, we must report that the programmer on this project is the first author of the present paper, and that he had no prior experience in SAC programming, was not associated with the SAC development team (while the other authors belong to it) and therefore was a good model of a brave 'computational scientist', willing to learn a new language. This was helped further by the fact that the author in question *is* a computational scientist by training (up to and including

```
inline
  double[*]
pent( double[.] p,
      double[.] q,
      double[.] bet,
      double[.] den,
      double[.] a,
      double[.] f)
{
  n = shape( a)[0];
  u = genarray( shape( f), undef);
  u[0] = f[0] * bet[0];
  u[1] = ( den[1] * u[0] - f[1]) * bet[1];

  for ( i=2; i<n; i++) {
    u[i] = ( a[i] * u[i-2] + bet[i] * u[i-1] - f[i]) *
           den[i];
  }
  u[n-2] = u[n-2] + p[n-2] * u[n-1];

  for ( i=n-3; i>=0; i--) {
    u[i] = u[i] + p[i] * u[i+1] + q[i] * u[i+2];
  }

  return( u);
}
```

Fig. 2. The main part of the pentdiagonal solver written in SAC

the doctoral level), with an established research record in this area. Hence the experiment in SAC coding should be considered relevant, if only small-scale. There is, of course, a slight inadequacy in that the author in question, while not being familiar with SAC at the start of the experiment, had taught various undergraduate subjects pertaining to functional programming, and so cannot be considered totally unfamiliar, even though a conscious effort was made to approach the task with a completely open, pragmatically driven mind.

Nevertheless, the first experience to be reported is that

Programming in SAC *does not require any re-tuning of the application programmer's mental skills.*

Indeed, as the code displayed so far suggests, the programmer only uses very familiar language features:

- *definitions*, perceived as assignments;
- *data-parallel definitions*, encoded as so-called *with-loops*, but which feel almost like normal elementwise assignment found in FORTRAN. The difference, while profound on the conceptual level, is superficial for an applications programmer.

```
inline
  double[*]
simps( double[*] f,
       double h)
{
  r = genarray( shape( f), undef);
  n = shape( f)[0];
  r = with (i)
       ([0]<=[i]<=[0]) :   (11.0*f[0]+14.0*f[1]-f[2])/24.0;
       ([1]<=[i]<=[n-2]) : (f[i-1]+4.0*f[i]+f[i+1])/3.0;
      genarray( [n], 0.0);

  rs = r[2]; r[2] = r[1]; r[1] = r[0]; r[0] = 0.0;

  for ( i=3; i<=n-1; i++) {
    x = r[i];
    r[i] = r[i-2] + rs;
    rs = x;
  }

  return( r * h);
}
```

Fig. 3. The integrator

– *functions* which, due to the availability of multiple results, feel more like FORTRAN procedures with the input and output parameters neatly separated out.

The conclusion is that SAC does not frighten off a computational scientist to the extent that fully-fledged functional languages would. There are plenty of familiar features in SAC, presented in a very slight guise, making the whole concept of SAC totally nonthreatening. More importantly, the following features of functional languages do *not* play any role in SAC, namely

– recursive functions and recursive data structures. Indeed, while the underlying mechanisms might be recursive, the appearance of the code is data-parallel (with-loops) and iterative/recurrent (for-loops). We did not use any of the generally recursive mechanisms of the functional paradigm.
– higher-order functions. These would be the main kind of "glue" in mainstream functional programming, and would normally present a considerable difficulty to a computational scientist, especially where cost intuitions are essential. A SAC applications programmer does not make any use of these at all.

To summarise, the reason why a computational scientist would find SAC usable are the absence of fundamentally unfamiliar concepts and the presence of familiar ones albeit in a somewhat unusual form.

The lack of control flow does not preclude "update mentality", thanks to the single-assignment rather than non-assignment semantics and terminology, but requires the programmer to be aware of the two major programming modes: data-parallel, via with-loops, and recurrent, via for-loops.

The programmer in this experiment felt acutely aware of recurrences. Indeed, the code shows the importance of recurrent definitions (and their explicit representations) quite convincingly. The integration function, the pentdiagonal solver and even the main computational scheme are all recurrent as well as being data-parallel. The programmer was assured by the other authors that the for-loop is translated efficiently by the SaC compiler, so recurrences need not be avoided. Equally, the programmer was continually aware of the data-parallelism of SaC constructs. In approaching those, the most important feature turned out to be rank subtyping, which allowed arrays to be represented in lower dimensions and consistently used in higher dimensions, as mentioned in the previous Section. This simplified testing as well as making the code unusually flexible.

Substitutional nature of SaC *definitions positively encourages the programmer to introduce as much notation as may be required to achieve readability and expressiveness.*

Under normal circumstances, the programmer is wary of extra variables in a program, as these normally cause additional memory allocation and, more importantly, additional memory cycles, synchronisation (if multithreading is used), cache conflicts, etc. So one's instinct would be to only use scalar "work" variables when formulae start to get too large. This applies even more to the use of functions, since the machinery of local variables and parameter-passing inflicts additional costs.

SaC, on the other hand, allows the programmer to forget such concerns completely. Indeed, the programmer in this experiment was assured by the SaC team that any variables defined in a function will be completely transparent: data will be "pulled through" them with no additional memory allocation or synchronisation being at all necessary. The same applies to inlined functions. They are completely transparent to the code generator of SaC, so it can be safely assumed that such functions act merely as substitutions at the source level, both semantically and efficiency-wise. The programmer has found that to be very useful.

It should be mentioned that the substitutional nature of variable definitions in SaC liberates the programmer from the duty to assiduously declare every variable that he may for any reason wish to introduce. Having to declare every variable is seen as a virtue in the imperative world: a modern Fortran programmer writes a proud "IMPLICIT NONE" in each module to prevent the compiler from using default types. However if the whole point of a variable is to denote a chunk of unwieldy expression which happens to have an application meaning, then there is no reason why that denotation must be fully attributed and potentially hold memory. It was a refreshing experience to use SaC variables

as pure notation, not memory address synonyms, for which the functional style ought to be credited.

5 FORTRAN Blues

For our performance studies, we have re-implemented the numerical method in FORTRAN 90/95 as a basis for comparison. One must note that writing the same code in FORTRAN was neither easy nor convenient. The main problem was that the whole rank structure of the algorithm had to be reconsidered. While, theoretically, rank polymorphism is available to the user of FORTRAN 95, in practice this is severely impeded by the total lack of rank subtyping. It turned out to be impossible to define a function that takes an argument of a higher rank and treats it as a uniform collection of lower-rank array components to be processed componentwise. The only exception is so-called elementwise functions

```
function simps(f,h)
   implicit none
   DOUBLE PRECISION,intent(in),dimension(0:XM-1,0:YM-1)::f
   DOUBLE PRECISION, intent(in) :: h
   DOUBLE PRECISION, dimension(0:XM-1,0:YM-1) :: simps
   DOUBLE PRECISION :: rs, w

   integer :: i,j

   do j=0, YM-1
      simps(0,j) = (11*f(0,j)+14*f(1,j)-f(2,j))/24*h
      do i=1,XM-1
        simps(i,j) = (f(i-1,j)+4.0*f(i,j)+f(i+1,j))/3*h
      end do
   end do

   do j=0, YM-1
    rs=simps(2,j)
    simps(2,j)=simps(1,j)
    simps(1,j)=simps(0,j)
    simps(0,j)=0

    do i=3,XM-1
     w=simps(i,j);
     simps(i,j)=simps(i-2,j)+rs
     rs=w
    end do
   end do

end function
```

Fig. 4. The FORTRAN version of the Simpson integrator

```
subroutine prepent(a,b,c,d,e,P,Q,BET,DEN)
  implicit none
  DOUBLE PRECISION,intent(in),dimension(0:XM-1)::a,b,c,d,e
  DOUBLE PRECISION,intent(out),dimension(0:XM-1)::P,Q
  DOUBLE PRECISION,intent(out),dimension(0:XM-1)::BET,DEN

  BET(0) = 1/c(0)
  P(0) = -d(0)*BET(0)
  Q(0) = -e(0)*BET(0)

  BET(1) = -1/(c(1)+b(1)*P(0))
  P(1)   = (d(1)+b(1)*Q(0))*BET(1)
  Q(1)   = e(1)*BET(1)
  den(1) = b(1)

  do i=2, XM-1
   BET(i)=b(i)+a(i)*P(i-2);
   DEN(i)= -1.0/(c(i)+a(i)*Q(i-2)+BET(i)*P(i-1));
   P(i)=(d(i)+BET(i)*Q(i-1))*DEN(i);
   Q(i)=e(i)*DEN(i);
  end do
end subroutine
```

Fig. 5. Elimination in FORTRAN

that can apply themselves to scalar components. The design ploy referred to earlier, when a function was defined on 1d arrays and applied to 2d arrays implicitly along the lower dimension, is not possible in FORTRAN.

We consequently had to opt for a fixed-rank design, and insert explicit DO-loops in the code which merely spanned the ranges of unprocessed index variables. We could have used data-parallel expressions with explicit array sections, but felt that that would obfuscate the algorithm even more than the extra indices. Figure 5 shows the FORTRAN version of the integrator, where all the j-loops had to be inserted into a code otherwise very similar to the one in Fig. 3.

The two parts of the linear solver had to be treated differently: the elimination stage was programmed as rank 1, see Fig. 5 whilst the back-substitution stage was made explicitly two-dimensional, Fig. 6.

On the positive side, the code is remarkably close to SAC, which demonstrates how low the barrier to the functional method would be for anyone involved in mainstream numerical computing. Such a programmer would only need to unlearn a few reflexes (avoidance of notation, variable declarations for nonessential objects, etc.) and perhaps learn a few SAC library functions.

6 Performance

We have measured the runtime of both FORTRAN and SAC versions of the porogram for varying problem sizes on three platforms: Intel XEON/Linux, AMD Athlon 64/Linux and Sun UltraSPARC/Solaris.

```
  function pent(p,q,bet,den,a,f)
2  implicit none
   DOUBLE PRECISION, intent(in), dimension (0:XM-1) :: p
4  DOUBLE PRECISION, intent(in), dimension (0:XM-1) :: q
   DOUBLE PRECISION, intent(in), dimension (0:XM-1) :: bet
6  DOUBLE PRECISION, intent(in), dimension (0:XM-1) :: den
   DOUBLE PRECISION,intent(in),dimension(0:XM-1,0:YM-1)::f
8  DOUBLE PRECISION, dimension (0:XM-1,0:YM-1) :: pent

10 do j=0,YM-1
      pent(0,j)=f(0,j)*bet(0)
12    pent(1,j)=(den(1)*pent(0,j)-f(1,j))*bet(1)
   end do
14
   do i=2, XM-1
16  do j=0,YM-1
    pent(i,j)=(a(i)*pent(i-2,j)+bet(i)*
18              *pent(i-1,j)-f(i,j))*den(i)
    end do
20 end do
   do j=0,YM-1
22  pent(XM-2,j)=pent(XM-2,j)+p(XM-2)*pent(XM-1,j)
   end do
24 do i=XM-3,0,-1
    do j=0, YM-1
26  pent(i,j)=pent(i,j)+p(i)*pent(i+1,j)+q(i)*pent(i+2,j)
   end do
28 end do
  end function pent
```

Fig. 6. Back substitution in FORTRAN

For the Intel and AMD processors, the FORTRAN code was compiled using the Intel Fortran Compiler (or `ifort` for short) version 9.0. For the SAC program, the current research compiler `sac2c` v1.00-alpha has been used. To yield comparable results, the Intel C Compiler (or `icc` for short) version 9.0 served as the back-end compiler for `sac2c`. For both Intel compilers the `-fast` option was specified to switch on any speed optimisations. Since the `-fast` option is currently not supported for AMD Athlon processors, a lesser option `-O3` was used when compiling for these.

The Sun UltraSPARC binaries were built using the Sun Studio Fortran compiler (or `sfort` for short) version 9.0. Again, for the sake of comparability, the Sun Studio C compiler (or `scc` for short) version 9.0 was used as the back-end compiler of `sac2c`. For both compilers, the `-fast` option was employed to obtain optimised binaries.

For all platforms the run-times of both implementations were measured in three consecutive runs and the average value was used. Values with a deviation higher than 5% were not considered.

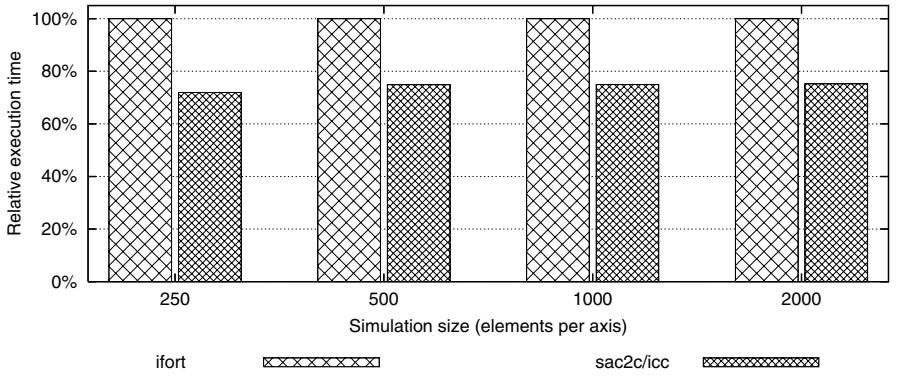

Fig. 7. Relative runtime of the two KPI implementations on a dual Intel XEON 3.0GHz machine. The runtime of the FORTRAN implementation is used as the base value.

Figure 7 shows the measured run-times on a dual Intel XEON 3.0GHz machine running Red Hat Enterprise Linux. The run-times of the SAC implementation are given relative to the FORTRAN run-times which serve as base values. The problem size is given in elements per axis of the data array which is the largest array size used within the algorithm. The results show that the SAC implementation outperforms the FORTRAN version by about 25%, despite its higher level of abstraction.

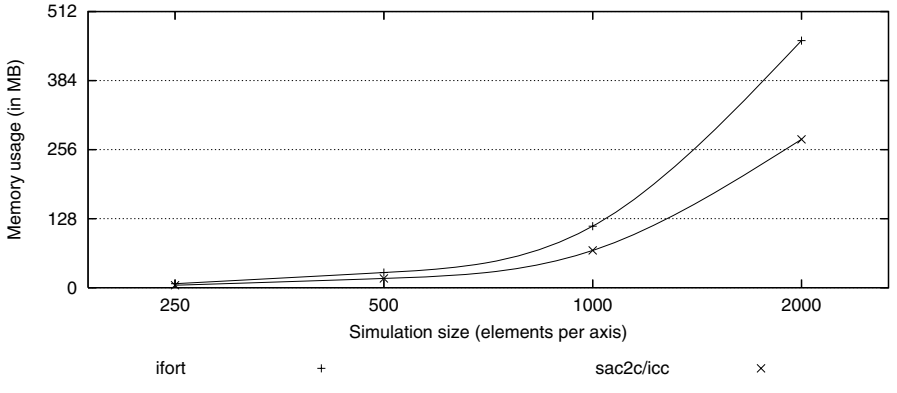

Fig. 8. Heap usage of the SAC and FORTRAN implementation in MB, measured on the Intel XEON machine

To find the reasons of the runtime advantage of the SAC implementation on the Intel XEON platform, we have measured the heap usage of both implementations for each problem size. Figure 8 gives the details. Obviously, the SAC implementation has a smaller memory footprint as the FORTRAN version, irrespective of the given problem size. A closer analysis reveals that the SAC

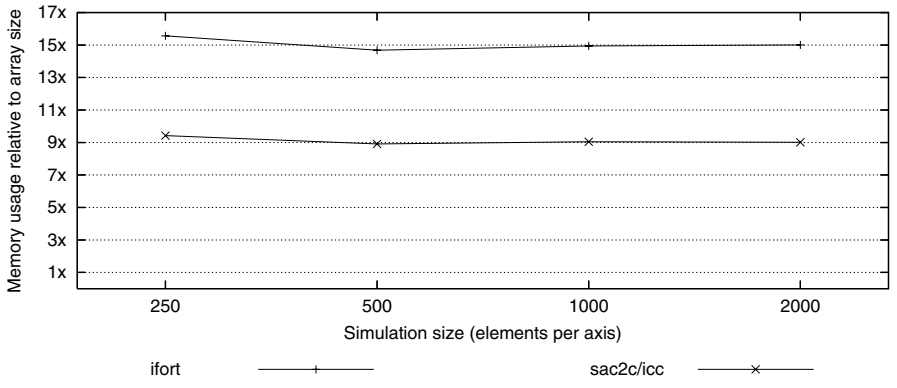

Fig. 9. Heap usage on the Intel XEON machine in multiples of data array size

heap usage is constantly 40% below the FORTRAN heap usage. This points to the SAC version of KPI handling memory reuse more efficiently and therefore requiring fewer simultaneous copies of the data array or intermediate arrays. To strengthen this assumption, we have calculated the overall heap usage in terms of multiples of the data array size. The results for both implementations of KPI are presented in Fig. 9.

The heap size of the FORTRAN implementation turns out to be about 15 times the size of the field array u. Given the declaration of 14 auxiliary arrays within the FORTRAN source code, this suggests that the FORTRAN compiler did not attempt to optimise array allocation. It seems that all arrays were allocated statically exactly as they have been declared by the programmer.

On the other side, the heap usage of the SAC version of KPI is approximately 9 times the size of the field array, despite liberal use of array expressions associated with auxiliary variables.

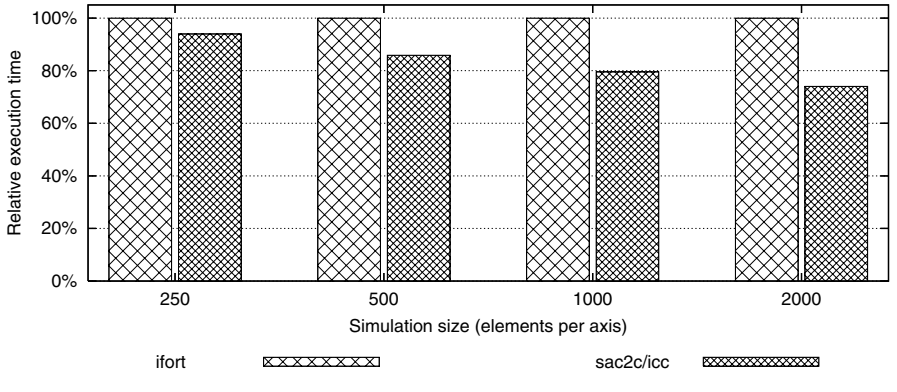

Fig. 10. Relative runtime of the two KPI implementations on a AMD Athlon 64 2.0GHz machine. The runtime of the FORTRAN implementation is used as the base value.

As a second benchmarking platform, an AMD Athlon 64 2.00GHz machine running SuSE Linux was used. Figure 10 gives the measured run-times. Similar to the results for the Intel XEON machine, the SaC implementation outperforms the Fortran version of KPI. Note that the advantage of the SaC version increases from about 7% for small problem sizes to 25% for a 2000 ×2000 data array.

This gave rise to the question whether low memory usage and high locality are more important to achieve good run times on the AMD machine than pure code efficiency. To investigate this further, we enabled a more aggressive version of With-loop Scalarisation [13] for the `sac2c` compiler to allow for optimisations that duplicate code to achieve higher locality and lower memory usage. Figure 11 shows the measured run times. The improvement is at least 10% for all tested problem sizes.

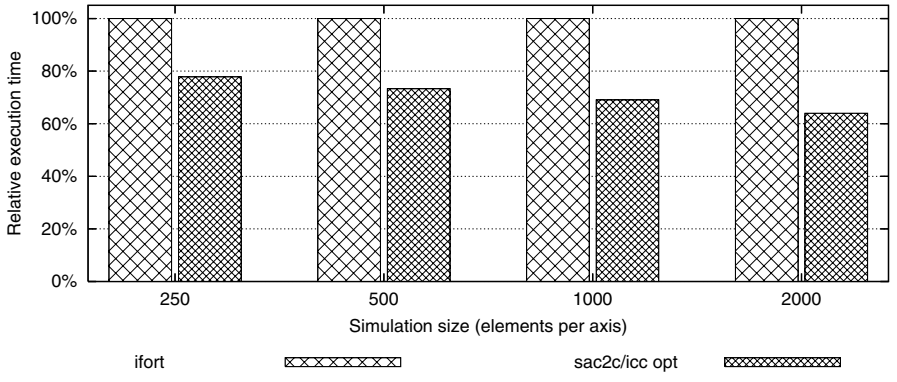

Fig. 11. Relative run time of the two KPI implementations on a AMD Athlon 64 2.0GHz machine. The SaC version was explicitly optimised for memory usage by the compiler. The runtime of the Fortran implementation is used as base value.

As our final benchmarking platform we used a SunFire 15k equipped with 72 UltraSparc III processors running at 900MHz under Sun Solaris. Figure 12 presents the measured run times. As on the AMD Athlon machine, we have measured two versions of the SaC implementation. The `sac2c/scc` version was compiled using the default settings of the `sac2c` compiler, whereas for the `sac2c/scc opt` version the more aggressive optimisations have been enabled.

To our surprise, the performance figures on the SunFire platform tell a different story compared to the cases discussed so far. For the conservatively optimised version of the SaC implementation, the Fortran version is between 50% and a factor of 2 faster. The more aggressively optimized version of the SaC implementation comes far closer to the Fortran version and even exceeds it in speed for large problem sizes. Compared to the results for the AMD Athlon machine, the differences between the two SaC versions is more striking. To investigate whether the performance difference is due to the better heap management of the

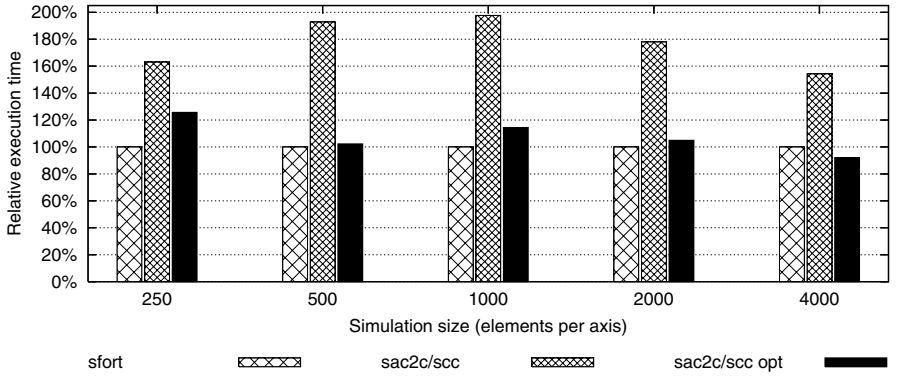

Fig. 12. Relative runtime of the two KPI implementations on a SunFire 15k. The SAC version denoted as `sac2c/icc opt` was explicitly optimised for memory usage by the compiler. The runtime of the FORTRAN implementation is used as the base value.

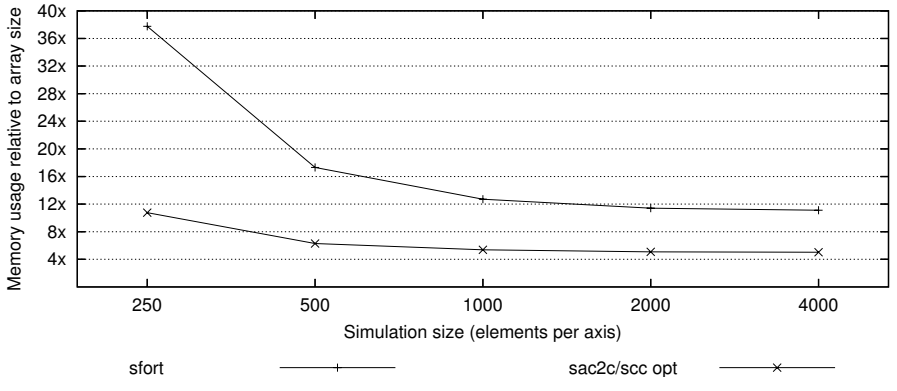

Fig. 13. Heap usage on the SunFire 15K in multiples of data array size

SAC compiler, we measured the heap usage of the aggressively optimised SAC version and the FORTRAN version. The results are given in figure 13. Here as well, we calculated the heap usage in multiplies of the size of the data array.

As the Figure shows, the heap usage of the SAC implementation converges at 5 times the data array size, whereas the FORTRAN implementation uses about eleven times the data array size. The huge difference in memory consumption for small problem sizes can be explained by memory used for initial setup.

An detailed investigation of why the Sun Studio Fortran compiler yields better runtime results while using more heap space would require close inspection of the object code and is as such beyond the scope of this paper. Due to the superscalar nature and large cache sizes of the SPARC processors being used, pipeline and cache effects may have a large impact on the runtime performance. As the SAC compiler does not generate machine code directly but instead uses the Sun C compiler as back end, pipeline and cache optimisations are out of its reach.

```
int main()
{
  dx = 0.1; dy = 0.1; dt = 0.0002;
  n = 500; m = 400; x0 = 15.0;
  alpha = [ 8.0/15.0, 2.0/15.0, 1.0/3.0] * dt;
  gamma = [ 8.0/15.0, 5.0/12.0, 3.0/4.0] * dt;
  rho = [ -17.0/60.0, -5.0/12.0] * dt;
  eps = alpha / ( 4.0*dx*dx*dx);
  u = with (ij)
        (. <= [i,j] <= .) : soliton( dx*tod( i) - x0,
                                     dy*tod( i-199));
      genarray( [ n, m], undef);
  a = genarray( [ n], -eps);
  a[0] = 0.0; a[1] = 0.0;
  a[n-2] = 2.0*a[n-3]; a[n-1] = a[n-2];
  a = { [i,j] -> a[j,i]};
  b = genarray( [n], 2.0*eps);
  b[0] = 0.0; b[1] = -b[2]; b[n-2] = 3.0*b[n-3];
  b[n-1] = 2.0*b[n-3]; b = { [i,j] -> b[j,i]};
  c = genarray( [n], 1.0);
  c[0] = c[0]+2.0*eps; c[1] = c[1]+6.0*eps;
  c[n-2] = c[n-2]-6.0*eps; c[n-1] = c[n-1]-2.0*eps;
  c = { [i,j] -> c[j,i]};
  d = genarray( [n], -2.0*eps);
  d[0] = 2.0*d[2]; d[1] = 3.0*d[2];
  d[n-2] = -d[n-3]; d[n-1] = 0.0; d = { [i,j] -> d[j,i]};
  e = genarray( [n], eps);
  e[0] = 2.0*e[2]; e[1] = e[0];
  e[n-2] = 0.0; e[n-1] = 0.0;
  e = { [i,j] -> e[j,i]};
  p, q, bet, den = prepent( a, b, c, d, e);
  out = 0;
  for ( iter=1; iter<100000; iter++) {
    out = display( u, iter, out);
    Nubase = N( u, dx, dy);
    f = u + gamma[0]*Nubase +
            alpha[0]*0.5*L(u,dx);
    u=pent( p[0], q[0], bet[0], den[0], a[0], f);
    f = u + gamma[1]*N( u, dx, dy) +
        rho[0]*Nubase + alpha[1]*0.5*L(u,dx);
    u = pent( p[1], q[1], bet[1], den[1], a[1], f);
    f = u + gamma[2]*N( u, dx, dy) +
        rho[1]*Nubase+alpha[2]*0.5*L(u,dx);
    u = pent( p[2], q[2], bet[2], den[2], a[2], f);
  }
  return( out);
}
```

Fig. 14. The main program of the SaC implementation

The conclusion to be drawn from these results is twofold. First of all, we have shown that the SAC compiler is capable of creating binaries in the same runtime league as two industrial-strength FORTRAN compilers. Secondly, we have seen that whether a FORTRAN implementation or SAC implementation yields better run times depends on the interplay of the SAC optimisations, the chosen C compiler and the executing machinery. To what extent the choice of these can be automated remains unclear and requires further research.

7 Conclusions

The results of a study of application programming in Single Assignment C has been presented. We have discussed the various design issues, principles and lessons arising from a programming exercise with a fairly mainstream equation, using several component methods: a linear solver, a Simpson space integrator and a Runge-Kutta time integrator. We have found Single Assignment C well-suited as a tool for developing numerical applications, especially when extensibility is required for future-proofness. We also found that the resulting code, although more flexible and easier to write than conventional FORTRAN, was not dramatically different in appearance, from which we conclude that SAC should present a low learning barrier to a busy computational scientist.

Finally we have contrasted the SAC code performance with that of an equivalent FORTRAN code using more than one compiler of commercial strength. We did that in order to establish whether the computational scientist might be discouraged from using the proposed methodology by unsatisfactory run-time efficiency, which could result from a liberal use of the functional programming method. The performance data we have obtained dispel this concern. In most cases they show SAC advantage in both speed and space utilisation thanks to a deeper level of optimisation that the SAC compiler is capable of.

Future work will focus on diversifying the benchmark code base by including component methods such as Monte-Carlo, sparse matrix algebra, etc. while continuing to provide whole application examples and supporting performance studies. The ultimate goal is to create a body of evidence for the advocacy of the functional method for computational science as well as the advocacy of the specific array manipulation methodology developed within the SAC project. Our hope is that this will help to convince the computation sector to adopt those methods and techniques in large-scale numerical modelling.

References

1. SAC development team: Single Assignment C. A definitive web site. (`http://www.sac-home.org`)
2. Grelck, C., Scholz, S.B.: Towards an efficient functional implementation of the nas benchmark ft. In: Proceedings of the 7th International Conference on Parallel Computing Technologies (PaCT'03), Nizhni Novgorod, Russia, LNCS 2763, Springer-Verlag (2003) 230–235

3. Hammes, J., Sur, S., Bohm, A.P.W.: On the effectiveness of functional language features: NAS benchmark FT. Journal of Functional Programming **7** (1997) 103–123
4. Serrarens, P.: Implementing the Conjugate Gradient Algorithm in a Functional Language. In Kluge, W., ed.: Implementation of Functional Languages, 8th International Workshop, Bad Godesberg, Germany, September 1996, Selected Papers. Volume 1268 of LNCS. Springer (1997) 125–140
5. Frens, J., Wise, D.: Auto-Blocking Matrix-Multiplication or Tracking BLAS3 Performance from Source Code. In: Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Volume 32(7) of SIGPLAN Notices. ACM (1997) 206–216
6. E.A.Kuznetsov, Musher, S., Shafarenko., A.: Collapse of acoustic waves in dispersive media. In Sagdeev, R., ed.: Nonlinear and Turbulent Processes in Physics. Harward Academic Publishers (1984) 335–349
7. Kadomtsev, B.B., Petviashvili, V.I.: On the stability of solitary waves in weakly dispersive media. Sov. Phys. Dokl **15** (1970) 539–541
8. Singh, S., Honzawa, T.: Kadomtsevpetviashivili equation for an ion-acoustic soliton in a collisionless weakly relativistic plasma with finite ion temperature. Physics of Fluids B: Plasma Physics **5** (1993) 2093–2097
9. Gilbert, G.: The kadomtsev-petviashvili equations and fundamental string theory. Communications in Mathematical Physics **117** (1988) 331–148
10. Chen, X.N., Sharma, S.D.: Zero wave resistance for ships moving in shallow channels at supercritical speeds. J. Fluid Mech. **335** (1997) 305321
11. Lu, Z., Tian, E.M., Grimshaw, R.: Interaction of two lump solitons described by the kadomtsev-petviashvili equation. Technical report, Dept. Math. Sci., Loughborough University (2003)
12. Timmes, F.X.: A pentadiagonal linear equation solver. (`http://www.cococubed.com/code_pages/pent.shtml`)
13. Grelck, C., Scholz, S., Trojahner, K.: With-loop scalarization – merging nested array operations. In Michaelson, G., Trinder, P., eds.: Proceedings of IFL'03 (selected papers). Volume 3145 of LNCS., Springer-Verlag (2004) 118–134