Algorithms and Applications on Vector and Parallel Computers
H.J.J. te Riele, Th.J. Dekker and H.A. van der Vorst (Editors)

Multigrid methods for elliptic PDEs.

W.M. Lioen
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

After a brief introduction in multigrid methods we discuss some of the algorithmic choices in the NUMVEC [1] Library routine MGZEB (which is a highly vectorised multigrid code for the solution of linear systems resulting from the 7-point discretisation of general linear 2^{nd} order elliptic PDEs in two dimensions).
Since the relaxation process is the most expensive part of a multigrid iteration cycle, we adapted the datastructure to avoid Cyber 205 stride-problems when executing zebra relaxation.
After discussing the effects of vectorisation / choosing another datastructure, we will also have a glance at large problems on the Cyber 205.
The implementation is available in auto-vectorisable ANSI Fortran 77.

1. Introduction

In this paper we describe the algorithmic choices in the NUMVEC Library routine MGZEB, a highly vectorised multigrid solver for elliptic PDEs, and we discuss some of its experimental results. Documentation of this routine can be found in [11].

Based on (scalar!) operation-counts [5] two of the more promising variants of the multigrid algorithm were developed simultaneously: MGD1V by P.M. de Zeeuw [6,7,8,17], using ILU-relaxation [4], the autovectorisable version of MGD1 by P. Wesseling[16] and the algorithm presented in the present paper, using zebra relaxation.

The aim is to obtain a black-box linear system solver, written in autovectorisable ANSI Fortran 77, where the user remains unaware of the underlying multigrid algorithm. Some of the results were already presented in [6,7].

In section 2 we describe the class of problems, to be solved. In sections 3, 4, 5 we describe the general multigrid algorithm and the specific algorithmic choices in MGZEB. The structure of the Fortran implementation is given in section 6. Sections 7, 8 are devoted to vectorisation in standard Fortran and what can be considered as fair performance measurements. In section 9 some experimental results are presented. In section 10 we shall have a glance at large problems (larger than central memory) on the Cyber 205. Finally, in the last section we formulate some conclusions.

1. NUMVEC is a CWI library of NUMerical software for VECtor computers.in FORTRAN.

2. The problem

We consider the linear 2nd order elliptic PDE in two dimensions on $\Omega \subset \mathbb{R}^2$

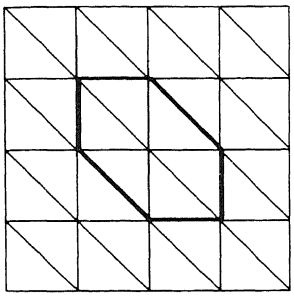
$$\sum_{i,j=1}^{2} a_{ij} \left[\frac{\partial}{\partial x_i}\right]\left[\frac{\partial}{\partial x_j}\right] u + \sum_{i=1}^{2} a_i \left[\frac{\partial}{\partial x_i}\right] u + a_0 u = f,$$

with variable coefficients and with boundary conditions on $\delta\Omega = \Gamma_N \cup \Gamma_D$

$$\left[\frac{\partial}{\partial n}\right] u + \alpha \left[\frac{\partial}{\partial s}\right] u + \beta u = \gamma \text{ on } \Gamma_N,$$

$$u = g \text{ on } \Gamma_D.$$

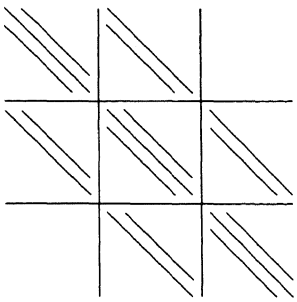
The coefficients are arbitrary smooth functions of x and should satisfy the ellipticity condition. If this equation on a rectangle Ω is discretised by means of a regular triangulation of the following form:



then the resulting discretisation

$$A_h u_h = f_h$$

can be a linear system with the following regular 7-diagonal structure.

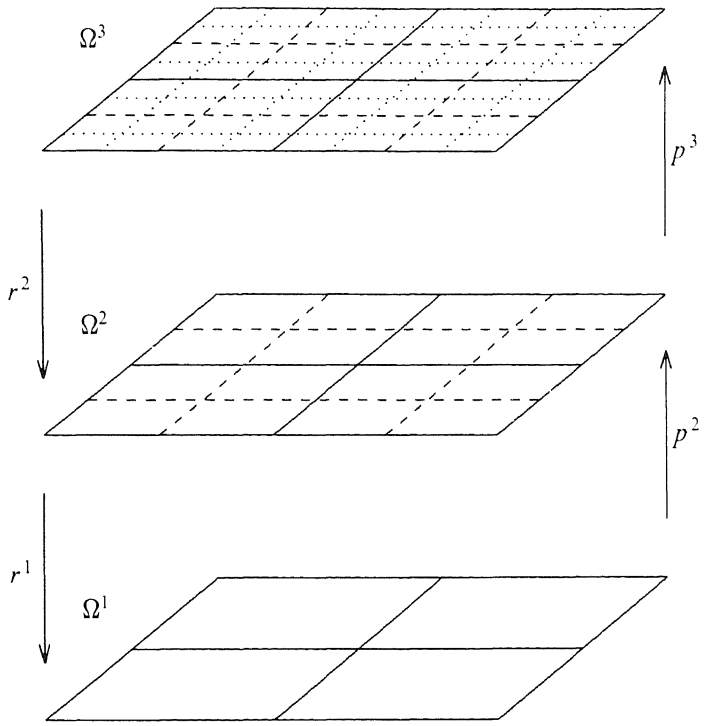


7-Point discretisation is the simplest discretisation which enables us to represent the cross-derivatives. It is the linear system that will be solved efficiently by means of a multigrid method [14].

On the rectangle Ω a sequence of uniform computational grids Ω^k, $k = 1\ (1)\ l$ is defined by

$$\Omega^k = \{(x_1, x_2) | x_i = x_{0_i} + jh_i^k,\ j = 0\ (1)\ 2^k\}\ ,$$

where: $h_i^{k-1} = 2h_i^k$, $k = l\ (-1)\ 2$ and $i = 1, 2$.



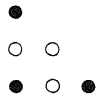
We denote the spaces of grid-functions on Ω^k by S^k. Prolongation and restriction operators are denoted by p^k and r^k

$$p^k: S^{k-1} \rightarrow S^k, \quad r^{k-1}: S^k \rightarrow S^{k-1}$$

and, if we do not want to specify the level, by $P: S_H \rightarrow S_h$, $R: S_h \rightarrow S_H$.

3. PROLONGATION, RESTRICTION AND COARSE-GRID-APPROXIMATION.

There are many possible choices for the prolongation and restriction operators [15]. In MGZEB for p^k a piecewise linear interpolation over the edges of a coarse triangulation is used.



p^k with ● a coarse-grid-point, ○ a fine-grid-point

In fact this is a natural choice in combination with a finite element discretisation with piecewise linear trial- and test-functions on the triangulation.

The corresponding restriction is the adjoint operator

$$r^{k-1} = \left(p^k\right)^T \text{ in the sense that}$$

$$(p^k u^{k-1}, v^k)_k = (u^{k-1}, r^{k-1} v^k)_{k-1} \quad \forall v^k \in S^k,$$

$$\text{with } (u^k, v^k)_k = \sum_{\Omega^k} u_{ij}^k v_{ij}^k \text{ , the usual inner product on } S^k,$$

which yields the following molecule for the restriction:

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \\ \frac{1}{2} & 1 & \frac{1}{2} \\ & \frac{1}{2} & \frac{1}{2} \end{bmatrix}.$$

Because this restriction is a weighted average over 7 points, the operators are also known as the 7-point restriction and prolongation.

For the solution of the discrete system $A^l u^l = f^l$ by a multigrid method, we also need the discrete operators A^k, $k = l-1\,(-1)\,1$ on the coarser grids Ω^k. Again several choices are possible. Since we want the user to supply only the matrix A^l and the right hand side f^l on the finest grid, the code has to generate the coarse-grid-operators by itself. For this purpose Galerkin approximation is used:

$$A^{k-1} = r^{k-1} A^k p^k, \; k = l\,(-1)\,2.$$

We call this Galerkin approximation because the following equality holds:

$$(A^k p^k u^{k-1}, p^k v^{k-1})_k = (A^{k-1} u^{k-1}, v^{k-1})_{k-1}$$

$$\forall\, v^{k-1} \in S^{k-1}.$$

Another motivation for the use of Galerkin approximation can be found in [16].

4. The multigrid algorithm

A comprehensive treatment of the multigrid method can be found in [14]. To briefly explain the multigrid algorithm here, we first introduce a two level algorithm, i.e. a multigrid algorithm with only two grids.
The two level algorithm is a relatively simple defect correction process:

- First a relaxation method such as Gauß-Seidel is applied to smooth the error. Such relaxations generally damp the high-frequency error components far more efficiently than the low-frequency error components.
- Secondly, the remaining (smoothed) error is transferred to the coarser grid by applying the restriction operator to the residual. Since the number of grid-points is much smaller on the coarser grid, the resulting system can be approximated far more efficiently, either by a direct solution process or -again- by applying a relaxation method. (On the coarser grid the error again has high-frequency components).

The correction thus found is transferred to the finer grid by means of the prolongation operator and added to the existing approximation.

- Finally, the result can again be smoothed by a relaxation method.

The former three steps are respectively called pre-relaxation, coarse-grid-correction and post-relaxation.

The two level algorithm is described in the following ALGOL-like fragment:

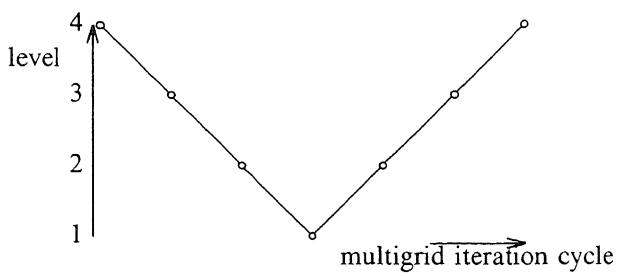
```
proc tla = (ref [,,] real A, ref [,] real u, f) void:
    begin
        to p do relax(A^2, u^2, f^2) od; #pre-relaxation#
        f^1 := r^1 (f^2 - A^2 u^2); #restriction of the residual#
        solve(A^1 u^1 = f^1); #solve directly#
        u^2 +:= p^2 u^1; #add prolongation of the correction#
        to q do relax(A^2, u^2, f^2) od #post-relaxation#
    end
```

Clearly, the two level algorithm can be used recursively to approximate the system on the coarser grid. This yields the multigrid (correction storage) algorithm described in the following ALGOL-like fragment:

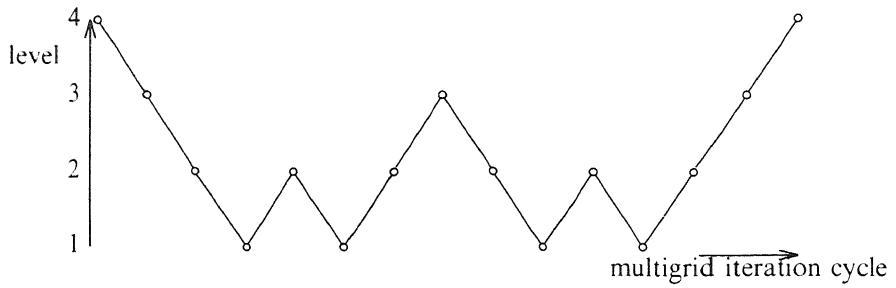
```
proc mgcs = (int level, ref [,,] real A, ref [,] real u, f) void:
    if level = 1 then
        solve(A^1 u^1 = f^1) #solve directly#
    else
        to p do relax(A^level, u^level, f^level) od; #pre-relaxation#
        f^(level-1) := r^(level-1) (f^level - A^level u^level); #restrict the residual#
        u^(level-1) := 0;
        #coarse-grid-correction: recursive application of mgcs#
        to σ do mgcs(level-1, A^(level-1), u^(level-1), f^(level-1)) od;
        u^level +:= p^level u^(level-1); #add prolongation of the correction#
        to q do relax(A^level, u^level, f^level) od #post-relaxation#
    fi
```

The integer values p, σ and q define the 'strategy' of the multigrid algorithm and respectively denote the number of pre-relaxations, coarse-grid-corrections and post-relaxations. For two strategies, we show how is switched between the different levels of discretisation.

For $l = 4$, $\sigma = 1$ we obtain a so-called V-cycle:



and for $l = 4$, $\sigma = 2$ we obtain a so-called W-cycle:



We see that most relaxation sweeps are performed on the lower levels. Taking into account the amount of work per iteration, we see that the total work on the lower levels is less than the relaxation work on the finest level.

5. The relaxation method

Now the multigrid algorithm has been described and the restriction, prolongation and coarse-grid-operators have been chosen; we still have to decide for a relaxation method.

Based on (scalar) operation counts [5] zebra relaxation is one of the promising ones. Also, for anisotropic problems, zebra relaxation is of special interest due to its excellent smoothing factors [14].

5.1. What is zebra relaxation?

Consider the system

$$Ax = b,$$

$$\sum_{j=1}^{n} a_{ij}x_j = b_i, \qquad i = 1\,(1)\,n.$$

The well-known (point-) Gauß-Seidel relaxation is defined as follows:

$$x_i = \frac{b_i - \sum_{j \neq i} a_{ij}x_j}{a_{ii}} \qquad \text{for } i = 1\,(1)\,n,$$

and block-Gauß-Seidel relaxation:

$$\text{solve:} \sum_{j \in \text{block}_l} a_{ij}x_j = b_i - \sum_{j \notin \text{block}_l} a_{ij}x_j \quad i \in \text{block}_l \quad \text{for } l = 1\,(1)\,\cdots\,(\otimes)$$

Note that after the l-th stage of the latter relaxation-process the residual on the l-th block vanishes, analogous to point-Gauß-Seidel relaxation, where for each point in turn the residual is zeroed.

Zebra relaxation is a special case of the block-Gauß-Seidel relaxation, where the blocks in the matrix correspond with the grid-lines in the mesh. Due to the

nature of the discretisation chosen, only information from neighbouring grid-lines is needed when one line is being relaxed. For line-Gauß-Seidel methods the systems ($\otimes$) reduce to tridiagonal systems.

A zebra relaxation sweep consists of two half-sweeps: relax the even lines first and then relax the odd lines, or conversely. To distinguish between both possibilities we call the relaxation even/odd or odd/even zebra relaxation.

The separate stages (corresponding with the lines) of each half-sweep can be carried out simultaneously: if we relax the even lines we only need information of the (neighbouring) odd lines and vice versa.

5.2. The choice even/odd- odd/even- zebra relaxation

In the multigrid-context one often has to calculate the restriction of the residual. The chosen 7-point restriction was determined by the following stencil:

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{2} & \\ \frac{1}{2} & 1 & \frac{1}{2} \\ & \frac{1}{2} & \frac{1}{2} \end{bmatrix}.$$

This means that in every coarse-grid-point the restriction is a weighted average of 3 points on a coarse-grid-line and 4 points on the two neighbouring fine-grid-lines. Let us have a closer look at the restriction after a zebra relaxation sweep has been performed.

As we have noticed before, the residual on the last relaxed lines has vanished, so, as a side-effect, we never have to calculate that part of the residual explicitly.

In the even/odd case, where we relax the odd lines last (i.e. the grid-lines that do not belong to a coarser grid) their contribution to the weighted average will be zero. This means, that the 7-point restriction effectively becomes a 3-point restriction, as opposed to the effective reduction to a 4-point restriction in the odd/even case:

$$\begin{bmatrix} \frac{1}{2} & 1 & \frac{1}{2} \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} \frac{1}{2} \\ 1 \\ \frac{1}{2} \end{bmatrix}.$$

From a computational complexity point of view, the choice is obviously in favour of even/odd zebra relaxation (the coarse-grid-lines first).

5.3. The choice x-zebra / y-zebra

Consider x-zebra, where the horizontal grid-lines correspond with the columns of two-dimensional arrays in the Fortran implementation:

- the computation of the right hand sides of the tridiagonal systems is done with stride 1 (line-wise)
- the simultaneous solution of those systems is done with stride $nx \times 2$ (skipping array-columns)

and y-zebra, vertical grid-lines corresponding with array-rows:

- the computation of the right hand sides of the tridiagonal systems is done simultaneous, with stride 2 (skipping array-rows)
- analogously, the simultaneous solution of those systems is also done with stride 2.

Consequently, because of our preference for stride 1, the most obvious choice is x-zebra relaxation.

5.4. Solving the resulting tridiagonal systems

With zebra relaxation we have to solve a tridiagonal system for every grid-line. Gauß-elimination turns out to be the most efficient solution method for small systems [10, 12]. For example on the STAR-100, the predecessor of the Cyber 205, Gauß-elimination is more efficient than cyclic reduction for systems up to order 160 [12].

Since we only have to solve tridiagonal systems on grid-lines and since the remaining recurrencies involved can be handled simultaneously, we choose for Gauß-elimination with storage of the LU-decompositions.

Consider the following tridiagonal system:

$$Au = \begin{bmatrix} a_1 & b_1 & & & 0 \\ c_2 & \ddots & \ddots & & \\ & \ddots & \ddots & \ddots & \\ & & \ddots & \ddots & b_{n-1} \\ 0 & & & c_n & a_n \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \\ u_n \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \\ \\ f_n \end{bmatrix} = f$$

We can write A as the product of a lower- and an upper- triangular matrix:

$$A = LU = \begin{bmatrix} 1 & & & & 0 \\ l_2 & \ddots & & & \\ & \ddots & \ddots & & \\ & & \ddots & \ddots & \\ 0 & & & l_n & 1 \end{bmatrix} \begin{bmatrix} d_1^{-1} & b_1 & & & 0 \\ & \ddots & \ddots & & \\ & & \ddots & \ddots & \\ & & & \ddots & b_{n-1} \\ 0 & & & & d_n^{-1} \end{bmatrix}$$

where

$$\left.\begin{aligned} d_1 &= 1/a_1 \\ l_i &= c_i d_{i-1} \\ d_i &= 1/(a_i - l_i b_{i-1}) \end{aligned}\right\} \quad i = 2\,(1)\,n.$$

The tridiagonal system is solved by a forward substitution

$Lv = f$ of the following form:

$$\begin{aligned} v_1 &= f_1 \\ v_i &= f_i - l_i v_{i-1}, \quad i = 2\,(1)\,n, \end{aligned}$$

followed by a backward substitution

$Uu = f$ of the form:

$$u_n = v_n d_n$$
$$u_i = (v_i - u_{i+1} b_i) d_i, \quad i = n-1\ (-1)\ 1.$$

Storing the reciprocal of the diagonal elements during the decomposition stage, we avoid an expensive division during the backward substitution.

The line-wise LU-decomposition contains a recursion and is therefore not immediately vectorisable. However, the quantities $c_i b_{i-1}$, $i = 2\ (1)\ n$, could be computed before the computation of the d_i. Also the computation of the l_i could be delayed until all the values of d_i have been obtained. Both these operations are vector multiplications and the line-wise LU-decomposition would look as follows:

$$\begin{aligned} t_i &= c_i b_{i-1} && i = 2\ (1)\ n \text{ (vector)} \\ d_1 &= 1 / a_1 \\ d_i &= 1 / (a_i - t_i d_{i-1}) && i = 2\ (1)\ n \\ l_i &= c_i d_{i-1} && i = 2\ (1)\ n \text{ (vector).} \end{aligned}$$

As mentioned before, the remaining recurrencies can be handled simultaneously due to the special ordering of grid-lines in a zebra relaxation sweep.

5.5. The choice of the datastructure.

Assume a Cyber 205 with 2 vector pipes and consider the following loop [2]:

```
      DO 10 I=1,N,2
10    U(I) = V(I) + W(I)
```

With vector length L = N/2 the total instruction timing (in clock-cycles) looks as follows:

scalar mode	17 +	19L
GATHER	2 × (39 +	5L/4)
ADD	51 +	L/2
SCATTER	71 +	5L/4
vector mode	200 +	4.25L

Equating the scalar and vector timing we note that the vectorised version is faster for L ⩾ 13. This is quite satisfactory, but when more GATHERS / SCATTERS are involved the break even point is often quite a bit higher. Opposed to a loop without a stride (thus only an ADD-instruction), we lose a factor 4 for short vectors (L = 2) and a factor 8.5! for long vectors (L → 65535).

Since the relaxation process is the most costly part of a multigrid cycle and since half the work of a zebra relaxation sweep is still done with a stride >1

when x-zebra is chosen, we have to look for another solution. Switching to y-zebra and using another datastructure i.e. renumbering the lines and grouping the even and the odd lines together, all the work inside the zebra relaxation can be done with a stride equal to 1.

6. The FORTRAN implementation of multigrid cycling

Although some algorithms can be described more elegantly in a recursive manner, it is often worthwhile to rewrite them in an iterative fashion. In our case the *counter* (in which the number of already performed coarse-grid-corrections on a certain level is kept) is the only local variable to be 'stacked'. Because we can easily differentiate between distinct cases, it is also possible to fully exploit the usage of cheaper residual, norm and restriction calculations after a zebra relaxation is performed as explained in section 5.2.

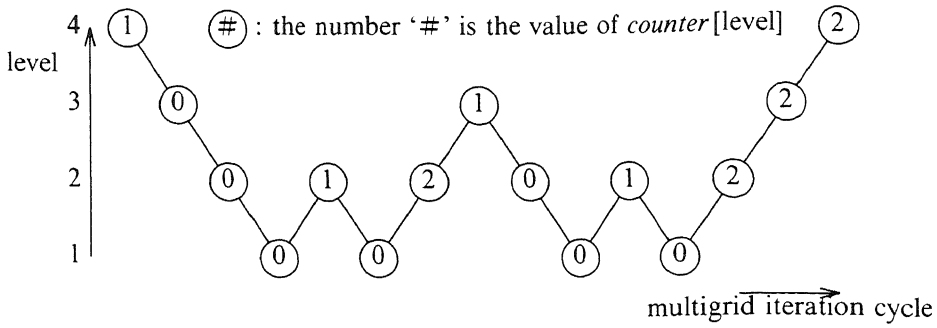
In order to avoid some Fortran details, the algorithm is again described in an ALGOL-like fragment:

```
proc cycles = (int levels, p, σ, q, ref [,,]real A, ref [,]real u, f) void:
    begin
        [levels] int counter;

        if p = 0 and σ>0 and levels>1 then
            if u^levels = 0 #in both cases normal residual#
                then f^(levels-1) := r^(levels-1) f^levels
                else f^(levels-1) := r^(levels-1) (f^levels - A^levels u^levels)
            fi
        fi;

        for mgit to maxit while ||f^levels - A^levels u^levels|| > tol
        do
            if levels = 1 or σ = 0
            then
                to p+q do relax(A^levels, u^levels, f^levels) od
            else
                counter[levels] := σ-1; #special initialisation#
                lev := levels;
    down:
                if p = 0
                then #down without pre-relaxation#
                    if lev ≠ levels then
                        #lev ≠ levels, so p = 0 implies q ≠ 0!#
                        d^lev := f^lev - A^lev u^lev #special residual#
                    fi;
                    if lev ≠ levels or mgit ≠ 1 then
                        # for lev = levels and mgit = 1
                          the restriction is already computed on
                          entry, otherwise p ≠ 0 or q ≠ 0 holds #
                        f^(lev-1) := r^(lev-1) d^lev #special restriction#
```

```
                fi;
                counter[lev − 1] := 0;
                for lvl from lev − 1 by −1 to 2
                do #(without pre-relaxation)#
                    f^(lvl−1) := r^(lvl−1) f^lvl; #normal restriction#
                    counter[lvl − 1] := 0
                od;
                u^1 := 0̄
            else #down with pre-relaxation#
                for lvl from lev by −1 to 2
                do
                    to p do relax(A^lvl, u^lvl, f^lvl) od;
                    d^lvl := f^lvl − A^lvl u^lvl; #special residual#
                    f^(lvl−1) := r^(lvl−1) d^lvl; #special restriction#
                    u^(lvl−1) := 0̄;
                    counter[lvl − 1] := 0
                od
            fi;

            #coarsest grid correction: also relaxation#
            lev := 1;
            to p + q do relax(A^1, u^1, f^1) od;
up:
            lev +:= 1;
            counter[lev] +:= 1;
            if p = 0 and counter[lev] = 1
                then u^lev := p^lev u^(lev−1)
                else u^lev +:= p^lev u^(lev−1)
            fi;
            to q do relax(A^lev, u^lev, f^lev) od;
            if counter[lev] < σ then goto down fi;
            if lev < levels then goto up fi
        fi;
        if q = 0 #no post-relaxation#
            then d^levels := f^levels − A^levels u^levels #normal residual#
            else d^levels := f^levels − A^levels u^levels #special residual#
        fi
    od
end
```

On the finest grid *counter*[*levels*] is initialised to $\sigma - 1$. Every time we go to a coarser grid, on invocation of σ coarse-grid-corrections, the counter on that level is initialised to zero. Every time we go to a finer grid, the counter on that level is incremented by one. If *counter*[l] $< \sigma$ we have to go down to the coarsest grid, otherwise we have to go up to the finest grid. This process terminates when we reach the finest grid, where, due to its special initialisation on entry, *counter*[*levels*] $= \sigma$ after incrementation.

The former is demonstrated in the following figure for $l = 4$, $\sigma = 2$: four levels with a W-cycle strategy:



7. SOME ASPECTS OF VECTORISATION IN STANDARD FORTRAN

This section reflects some of our experiences while writing MGZEB in autovectorisable standard Fortran. See chapter 9 of [1] or see[9] for a full treatise on vectorisation on the Cyber 205.

- Loop collapsing: in the following simplified loop nest only the inner loop is autovectorisable

```
      DIMENSION U(100,100), V(100,100), W(100,100)

      NX = 100
      NY = 100

      DO 10 J = 1, NY
      DO 10 I = 1, NX
   10 U(I,J) = V(I,J) + W(I,J)
```

  here loop collapsing can be accomplished by explicit overindexing:

```
      DO 10 I = 1, NX*NY
   10 U(I,1) = V(I,1) + W(I,1)
```

  A non-trivial example is the collapsing of the loops (over the grid-lines) occurring in the right-hand side computation of the tri-diagonal systems during the zebra relaxation in the adapted datastructure of MGZEB.

- In order to allow for the above mentioned loop collapsing, dummy array elements are added to the arrays containing the information of the odd grid-lines, in order to make their dimensions compatible with the arrays containing the information of the even grid-lines. (This instead of using sparse vectors on the Cyber 205, a highly non-standard, hardware supported, data-representation.)

- Periodic GATHER (and SCATTER analogously, think of the restriction and prolongation operators), can be written as

```
      DO 10 I=1,N
10    WORK(I) = U(2*I-1)
```

and will be recognised as such, by most vectorising compilers.

- On the Cyber 205, the infamous 'possible recurrencies' can be taken care of by aliasing or implicit equivalencing: passing the same actual argument to two dummy arguments. This possible recurrency occurs when the first index exceeds the maximum rowindex in the actual declaration, which is allowed within Fortran and therefore suspected by the Cyber 205 Fortran 200 compiler.
  The Cray CFT compiler assumes implicitly that the programmer avoids such a situation.

The Cray CFT compiler is also provided with compiler-directives to direct the vectoriser. In our opinion this is the most elegant solution to most problems mentioned in this section.

8. Measuring acceleration factors and MFLOPS

This section is devoted to what we consider fair performance measurements.

Writing autovectorisable programmes and scalar optimisation [13] do not always tally. On a vectorcomputer for example, one wants vectorisable inner loops and on a scalarcomputer one wants outer loops with the smallest iteration count. This example is demonstrated in the following obviously not vectorisable recursive loop:

```
      DIMENSION U(100,10)

      DO 20 J = 1, 10
        DO 10 I = 2, 100
          U(I,J) = U(I,J) + U(I-1,J)
10      CONTINUE
20    CONTINUE
```

where a simple exchange of loop indices is all that is needed to introduce the desired vector structure albeit with stride 100.

However, both versions are not optimal in scalar mode. It is possible to save LOAD/STORE-overhead by keeping the iterand in a register (a local scalar variable):

```
      DO 20 J = 1,10
        UIM1 = U(1,J)
        DO 10 I = 2, 100
          UIM1 = U(I,J) + UIM1
          U(I,J) = UIM1
10      CONTINUE
20    CONTINUE
```

From the examples mentioned above, it is clear that for fair measurement of acceleration factors we need to compare our timings with those of the most

efficient scalar version so:

fair acceleration factor = scalar optimised timing / vectorised timing,

and consequently:

fair MFLOP-rate = scalar operation count / vectorised timing.

9. Some experimental results

Since we are only interested in the vectorisation here and not in the numerical behaviour, we restrict ourselves to one single test problem. We solve the Poisson equation on the unit square with Dirichlet boundary conditions and the right hand side constructed in agreement with the exact solution $x(1-x)+y(1-y)$. In our example the boundary conditions are eliminated. A fixed 'saw-tooth' multigrid cycle strategy ($p = 0$, $\sigma = q = 1$) is used.

In the following tables we give CPU-times spent in various subroutines in runs with 10 multigrid cycles. Additionally, the average convergence factors in the iterative cycling are given.

machine	Cray 1S		Cyber 205 (2-pipes)	
levels	6	7	6	7
finest grid	65×65	129×129	65×65	129×129
convergence	0.232	0.218	0.232	0.218
RAP	0.033 (2.7)	0.085 (3.7)	0.022 (3.9)	0.054 (5.9)
DECOMP	0.006 (1.0)	0.023 (1.0)	0.010 (1.1)	0.040 (1.1)
ZEBRA	0.034 (4.0)	0.103 (5.0)	0.084 (1.8)	0.210 (2.8)
RESIDU	0.010 (4.9)	0.034 (5.6)	0.007 (6.4)	0.020 (9.0)
PROLON	0.008 (4.4)	0.022 (6.0)	0.013 (2.1)	0.032 (3.1)
RESTRI	0.004 (3.2)	0.009 (5.4)	0.009 (1.1)	0.022 (1.7)
VL2NOR	0.003 (4.3)	0.009 (5.3)	0.002 (4.0)	0.004 (8.3)
TOTAL	0.102 (3.4)	0.293 (4.4)	0.162 (2.2)	0.400 (3.3)
CYCLE	0.006 (3.8)	0.017 (5.2)	0.011 (2.1)	0.028 (3.2)

Table 9.1 CPU-times in seconds of the experimental programme MGEOZV (a version with the 'conventional' datastructure) run in vector-mode. Between parentheses the 'fair' acceleration by vectorisation.

We see that the Cray 1S does not suffer as much from strides as the Cyber 205: compare the acceleration factors 3.2, 5.4 and 1.1, 1.7 of the restriction for the Cray and the 205, respectively. The acceleration factors for the zebra relaxation are better than for the restriction, due to the fact that half the work can be done with stride 1.

Note, that on the Cyber 205, the relaxation consumes half the time (0.210 sec) of the total run time (0.400 sec) on a 129 × 129 grid.

In the next table we show the results after adapting the datastructure:

grid	65×65		129×129		257×257	
CHANGE	0.004		0.012		0.041	
RAP	0.021	(2.8)	0.050	(4.2)	0.147	(5.5)
DECOMP	0.002	(4.5)	0.004	(8.8)	0.013	(10.5)
ZEBRA	0.025	(5.7)	0.065	(8.4)	0.220	(10.2)
RESIDU	0.004	(10.8)	0.015	(11.2)	0.070	(9.6)
PROLON	0.016	(1.6)	0.036	(2.8)	0.094	(4.1)
RESTRI	0.015	(0.7)	0.033	(1.1)	0.083	(1.7)
VL2NOR	0.001	(9.0)	0.002	(16.5)	0.009	(14.3)
TOTAL	0.109	(3.0)	0.243	(4.9)	0.710	(6.6)
CYCLE	0.008	(3.2)	0.017	(5.3)	0.049	(7.2)

TABLE 9.2 CPU-times in seconds of the programme MGZEB (the version with the adapted datastructure) run in vector mode on a Cyber 205 (2-pipes)
Between parentheses the 'fair' acceleration by vectorisation.

Here appears a subroutine called CHANGE. This portable Fortran routine adapts the datastructure of the problem on entry and as we can see it only takes 0.041 sec on a 257 × 257 grid; this time is comparable to one multigrid iteration cycle.

We notice a few differences with the results given in table 9.1. The restriction suffers from vectorisation on small grids, see the acceleration factor 0.7 on the 65 × 65 grid. This loss is due to halving the vector length in the adapted datastructure.

Due to the absence of strides the zebra relaxation has excellent acceleration factors 5.7, 8.4 and 10.2! On a 129 × 129 grid the relaxation now only takes one fourth of the time of the total run, which in turn takes only 0.243 sec instead of 0.400 sec in the case of the 'conventional' datastructure.

The overall conclusion is, that using an adapted datastructure proves to be very profitable.

10. A GLANCE AT LARGE PROBLEMS ON THE CYBER 205

Although the results in this section strongly depend on the site's accounting system and are presented on a Cyber 205-611, the general idea should hold even for non-virtual memory machines, where the user has to take care of his own (explicit, more visible) I/O.

For readers unfamiliar with the Cyber 205, it is a so-called virtual memory machine, which means that virtual space (disk space, slow) is mapped to physical space (central memory, fast) during task execution. The process of copying code and data in and out of central memory is called paging or implicit I/O. The units transferred in and out are called pages. VSOS (the Cyber 205's

Virtual Storage Operating System) uses two page sizes, small and large pages. The large page size is always 128 blocks of 512 words (65536 words) and the small page size can be one, four, or sixteen blocks of 512 words. At SARA (the Academic Computing Centre Amsterdam) this installation parameter is 4 blocks of 512 words (2048 words).

A so-called SBU, a System Billing Unit, is a VSOS accounting unit (at SARA one CPU second corresponds with one SBU).

In the sequel we consider a 257×257 grid, in our case this is a 'large' problem (all data involved in this problem needs more than 1 Mword, which, in turn, is the current physical memory on the Cyber 205-611 at SARA; the maximal working set at the time of these measurements was 1600 blocks of 512 words).

In the following table, the problem is mapped on Large Pages and we iterate till $\|\text{residu}\| < 10^{-10}$

strategy p	σ	q	iterations needed	CPU (sec)	page faults SP	LP	jobcosts SBU's
0	1	1	20	1.44	636	408	49.3
0	2	1	11	1.85	566	235	31.5
0	3	1	11	7.18	547	244	47.4

As we can see W-cycle's ($\sigma = 2$) cost slightly more CPU-time than V-cycle's, but the total job-cost decreases significantly due to the reduced amount of relaxation sweeps on the finest grid (larger than central memory, so paging is involved).

Now perform 10 iterations with a fixed 'saw-tooth' cycle strategy ($p = 0$, $\sigma = q = 1$), but use scalar and vector compilations of MGZEB and different mappings of the problem

compilation mode	problem mapping	CPU (sec)	page faults SP	LP	jobcosts SBU's
scalar		7.79	4046	-	49.0
vector	(SP)	0.92	4053	-	42.1
vector	(LP)	0.84	507	219	28.2
In scalar mode it is also possible to map the problem on Large Pages!					
scalar	**(LP)**	7.71	32	203	28.7

Thus despite the increasing CPU-time the scalar version can be executed at the same job-costs due to the decreased number of page faults!

At the present 1 Large Page fault costs

$$0.1 \text{ SBU} \equiv 0.1 \text{ CPU sec} \equiv 76 \times 2^{16} \text{ clock-cycles, and}$$
$$0.5 \text{ sec real time} \equiv 381 \times 2^{16} \text{ clock-cycles!}$$

We see that, one can perform 381 × [the number of vectorpipes] vectorinstructions of maximal vectorlength in the same real time, so recomputing instead of storing intermediate (vector) results could be worthwhile.

11. Conclusions

We conclude that it is very well possible to write highly vectorisable code in standard Fortran (albeit a little bit tricky sometimes).

On the Cyber 205 the lack of compiler-directives to direct the vectoriser of Fortran 200 can be really painful (e.g. the possible recurrencies).

In order to obtain an optimal performance it can be necessary to adapt the datastructure of a programme to the specific machine's architecture.

If a certain loop-construct is only vectorisable at the cost of introducing page-faults then one should really consider whether the gain in performance is not outweighed by the extra costs of the page-faults.

Although some people do not care about the total job-costs and only want to obtain an optimal vector-performance ('Macho FLOP' people) it is more realistic to be interested in the real time (the turn-around time of the job).

Het zijn de programma's die het hem doen.

Acknowledgements

I would like to thank Piet Hemker and Paul de Zeeuw for their help, patience, critics and comments during all stages of this research.

Note

The code discussed in this paper can be obtained by sending a tape to the NUMVEC-Library manager, Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands.

References

1. Control Data Corporation (1981). *Cyber 200 Fortran Version 2 Reference Manual.*
2. Control Data Corporation (1982). *Cyber 205 User Guide.*
3. H. Foerster, K. Stueben, and U. Trottenberg (1981). Non-Standard multigrid techniques using checkered relaxation and intermediate grids, in *Elliptic problem solvers*, 285-300, ed. M. Schultz, Academic Press.
4. P.W. Hemker (1980). The incomplete LU-decomposition as a relaxation method in multigrid algorithms, in *Boundary and interior layers - Computational and asymptotic methods*, 306-311, ed. J.J.H. Miller, Boole Press.
5. P.W. Hemker (1982). On the comparison of line-Gauss-Seidel and ILU-

relaxation in multigrid algorithms, in *Computational and asymptotic methods for boundary and interior layers*, 269-277, ed. J.J.H. Miller, Boole Press.

6. P.W. HEMKER, R. KETTLER, P. WESSELING, and P.M. DE ZEEUW (1983). Multigrid methods: development of fast solvers, *Appl. Math. Comp.*, 13, 311-326.
7. P.W. HEMKER, P. WESSELING, and P.M. DE ZEEUW (1984). A portable vector-code for autonomous multigrid modules, in *PDE SOFTWARE: Modules, Interfaces and Systems*, 29-40, ed. B. Engquist & T. Smedsaas, North-Holland.
8. P.W. HEMKER and P.M. DE ZEEUW (1985). Some implementations of multigrid linear system solvers, in *Multigrid methods for integral and differential equations*, 85-116, ed. H. Holstein & D.J. Paddon, Oxford Press.
9. M.J. KASCIC JR. (1979). Vector Processing on the Cyber 200, *First published in the Infotech State of the Art Report "Supercomputers"*, Infotech International Limited.
10. J.J. LAMBIOTTE JR. and R.G. VOIGT (1975). The Solution of tridiagonal systems on the CDC STAR-100 computer, *ACM Transactions on Mathematical Software*, 1.4, 308-329.
11. W.M. LIOEN (1985). *NUMVEC FORTRAN Library manual, Chapter: Elliptic PDEs, Routine: MGZEB*, NM-R8518, Centre for Mathematics and Computer Science.
12. N.K. MADSEN and G.H. RODRIGUE (1976). *A comparison of direct methods for tridiagonal systems on the CDC STAR-100*, #UCRL-76993, Lawrence Livermore Laboratory.
13. M. METCALF (revised 1981). *Fortran program optimization*, Report CERN80-08, CERN.
14. K. STUEBEN and U. TROTTENBERG (1982). Multigrid methods: fundamental algorithms, model problem analysis and applications, in *Multigrid methods. Procs. Koeln-Porz, 1981. Lect. Notes in Math. 960*, 1-176, ed. W. Hackbusch & U. Trottenberg, Springer-Verlag.
15. P. WESSELING (1982). Theoretical and practical aspects of a multigrid method, *SIAM J. Sci. Stat. Comp.*3, 387-407.
16. P. WESSELING (1982). A robust and efficient multigrid method, in *Multigrid methods. Procs. Koeln-Porz, 1981. Lect. Notes in Math. 960*, 614-630, ed. W. Hackbusch & U. Trottenberg, Springer-Verlag.
17. P.M. DE ZEEUW (1986). *NUMVEC FORTRAN Library manual, Chapter: Elliptic PDEs, Routine: MGD1V and MGD5V*, NM-R8624, Centre for Mathematics and Computer Science.