# Parallelizing a Highly Vectorized Multigrid Code with Zebra Relaxation

Walter M. Lioen <walter@cwi.nl>

CWI, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

**Abstract**

*After a brief introduction in multigrid methods we discuss some of the algorithmic choices in MGZEB, a parallelized highly vectorized multigrid code for the solution of linear systems resulting from the 7-point discretization of general linear second order elliptic PDEs in two dimensions. We describe the minimization of the scalar operation count, the vector tuning on a vector-register machine and the parallelization of the already existing highly vectorized MGZEB code. At present we would use the same algorithm with the same scalar operation count on a scalar uni-processor. Finally, we discuss the overall parallel vector-performance using autotasking on a Cray Y-MP4/464.*

## 1: Introduction

In this paper we describe the algorithmic choices in MGZEB, a parallelized highly vectorized multigrid solver for elliptic PDEs, and we discuss some of its experimental results. Documentation of this routine can be found in [10].

Based on scalar operation-counts [4] two of the more promising variants of the multigrid algorithm were developed simultaneously: MGD1V by P.M. de Zeeuw [5-7, 20], using ILU-relaxation [3], the autovectorizable version of MGD1 by P. Wesseling [18] and MGZEB by the author of this paper [11], using zebra relaxation. Both variants were parallelized simultaneously: MGD1M, the multitasked version of MGD1V, by M. Louter-Nool [13] and MGZEB, presented in this paper.

The aim is to obtain a black-box linear system solver, written in autoparallelizable autovectorizable ANSI Fortran 77, where the user remains unaware of the underlying multigrid algorithm. Some of the numerical and vectorization results were already presented in [5, 6, 11].

In Section 2 we describe the class of problems, to be solved. In Section 3 we choose the prolongation, restriction and coarse-grid operators. We briefly describe the multigrid algorithm in Section 4. In Section 5 we select twisted-even/odd-y-zebra-relaxation as smoothing process for the multigrid algorithm on parallel vector computers. Section 6 is devoted to tuning existing vectorizable code on vector-register machines, like the Cray, and parallelizing in standard Fortran on a shared-memory multi-processor. In Section 7 some experimental results are presented. First we present the performance gain obtained by minimizing the scalar operation count and vector-tuning on a Cray Y-MP. Furthermore, we will discuss the effect of autotasking on a Cray Y-MP4/464. Finally, in the last section we formulate some conclusions.

## 2: The problem

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

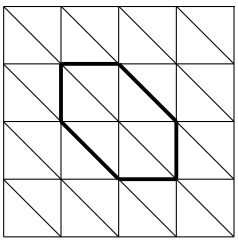
$$\sum_{i,j=1}^{2} a_{ij} \left(\frac{\partial}{\partial x_i}\right)\left(\frac{\partial}{\partial x_j}\right) u + \sum_{i=1}^{2} a_i \left(\frac{\partial}{\partial x_i}\right) u + a_0 u = f,$$

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

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

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

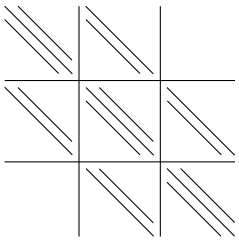
The coefficients are arbitrary smooth functions of $x$ and should satisfy the ellipticity condition. If this equation on a rectangle $\Omega$ is discretized by means of a regular triangulation of the following form:



then the resulting discretization

$$A_h u_h = f_h$$

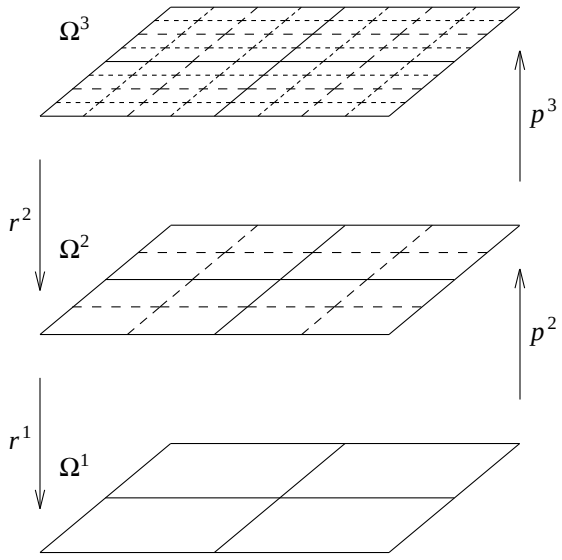
can be a linear system with the following regular 7-diagonal structure.



7-Point discretization is the simplest discretization which enables us to represent the cross-derivatives. It is the linear system that will be solved efficiently by means of a multigrid method [2, 15, 19].
On the rectangle $\Omega$ a sequence of uniform computational grids $\Omega^k$, $k=1\ (1)\ l$ is defined by

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

where the $h_i^k$, $i=1,2$ denote the meshwidths in horizontal and vertical direction and $h_i^{k-1} = 2h_i^k$, $k = l(-1)2$.




We denote the spaces of grid-functions on $\Omega^k$ by $S^k$

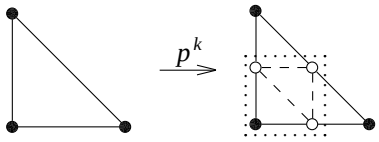
$$S^k = \{u^k : \Omega^k \to \mathbb{R}\}.$$

Prolongation and restriction operators are denoted by $p^k$ and $r^k$

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

## 3: Prolongation, restriction and coarse-grid-approximation

There are many possible choices for the prolongation and restriction operators [17, 19]. For a more detailed description of the operators chosen in this section we refer to [12].
In MGZEB for $p^k$ a piecewise linear interpolation over the edges of a coarse triangulation is used.




This is the natural choice in combination with a finite element discretization with piecewise linear trial- and test-functions on the triangulation.

The corresponding restriction is the adjoint operator

$$r^{k-1} = \left(p^k\right)^T$$

in the sense that

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

with $(u^k, v^k)_k = \sum_{\Omega^k} u_{ij}^k v_{ij}^k$, the usual inner product on $S^k$, which yields the following molecule for the restriction:

$$\begin{bmatrix} ½ & ½ & \\ ½ & 1 & ½ \\ & ½ & ½ \end{bmatrix}.$$

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 $\Omega^k$. 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:

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

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 [18, 19]. It can easily be shown that if $A^k$ has a general 7 (or fewer)-point structure, then $A^{k-1}$ has a 7-point structure. In [12] we present a new optimally efficient (scalar operation count, vectorizable and parallelizable) algorithm to construct the Galerkin coarse-grid approximations.

## 4: The multigrid algorithm

A comprehensive treatment of the multigrid method can be found in [2, 15, 19]. 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 relative 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 called pre-relaxation, coarse-grid-correction and post-relaxation, respectively. Clearly, the two level algorithm can be used recursively to approximate the system on the coarser grid. Furthermore, it is common practice to perform relaxations instead of directly solving the system on the coarsest grid; for a diffusion problem with pure Neumann boundary conditions it is even the only possibility. Assuming $level \geq 2$ this yields the following multigrid (correction storage) algorithm:

procedure mgcs($level$, $A^{level}$, $u^{level}$, $f^{level}$)
  Pre-relaxation:
  for $i = 1, \cdots, p$
    relax($A^{level}$, $u^{level}$, $f^{level}$)
  $f^{level-1} := \overline{r}^{level-1}(f^{level} - A^{level} u^{level})$
  $u^{level-1} := 0$
  Coarse-grid-correction:
  if $level > 2$ then
    Recursive application of mgcs:
    for $i = 1, \cdots, \sigma$
      mgcs($level-1$, $A^{level-1}$, $u^{level-1}$, $f^{level-1}$)
  else
    On coarsest grid, also relaxation:
    for $i = 1, \cdots, p+q$
      relax($A^1$, $u^1$, $f^1$)
  $u^{level} := u^{level} + p^{level} u^{level-1}$
  Post-relaxation:
  for $i = 1, \cdots, q$
    relax($A^{level}$, $u^{level}$, $f^{level}$)

The integer values $p$, $\sigma$ and $q$ define the 'strategy' of the multigrid algorithm and respectively denote the number of pre-relaxations, coarse-grid-corrections and post-relaxations. For $\sigma = 1$ we obtain so-called V-cycles and for $\sigma = 2$ we obtain so-called W-cycles.

Although some algorithms can be described more elegantly in a recursive manner, it is often worthwhile to rewrite them in an iterative fashion. It is even mandatory if we use Fortran. 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'. In [11] we give a detailed description of the iterative implementation of the multigrid correction storage algorithm. Because we can more easily differentiate between distinct cases in the iterative formulation, it is also possible to fully exploit the usage of cheaper residual, norm and restriction calculations after a relaxation is performed, as will be explained in the following section.

## 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 [4] zebra relaxation is one of the promising ones. Also, for anisotropic problems, zebra relaxation is of special interest due to its excellent smoothing factors [15].

Zebra relaxation is a special case of the line-Gauß-Seidel relaxation. A zebra relaxation sweep consists of two half-sweeps: first relax the even numbered lines and then relax the odd numbered lines, or conversely. To distinguish between both possibilities we call the relaxation even/odd or odd/even zebra relaxation. Due to the nature of the discretization chosen, only information from neighboring grid-lines is needed when a line is being relaxed. Using the old approximation on the neighboring grid-lines the 7-diagonal systems on the grid-lines reduce to tridiagonal systems (only the coupling on single grid-lines remains). Note that solving the resulting tridiagonal system on a grid-line by definition zeroes the residual on that grid-line; this is completely analogous to point-Gauß-Seidel relaxation, where for each point in turn the residual is zeroed.

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

### 5.1: Scalar operation count: even/odd-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:

$$\begin{bmatrix} ½ & ½ & \\ ½ & 1 & ½ \\ & ½ & ½ \end{bmatrix}.$$

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 neighboring 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:

$$\begin{bmatrix} ½ & 1 & ½ \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} ½ \\ 1 \\ ½ \end{bmatrix}.$$

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

### 5.2: Vector computers: y-zebra relaxation

Since we are solving multiple uncoupled tridiagonal systems, we can perform all operations simultaneously, thus vectorized, without introducing additional work. This technique of vectorizing a single dimension recursion is also known as *vectorizing across the problem*. All vector lengths equal the number of tridiagonal systems. Since we can get our vectorization for free, and because all other vectorizable and/or parallelizable tridiagonal system solving techniques introduce additional work, we will use Gaußian elimination, the technique with minimal scalar operation count. Note that for subsequent zebra relaxations, on a given level of discretization, only the right hand sides differ. This can be exploited by storing decompositions of the tridiagonal matrices. In [11] we describe the choice x-zebra / y-zebra in more detail, so we will only summarize the result here. Using another data structure 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. In order to allow 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. Clearly the performance gain is most significant on a Cyber 205. However, most (vector-) computers, including the Crays, benefit from a unitary-strided memory access. The price we had to pay was: the explicit change of the data structure on entry; halving of the vector-lengths in the restriction and prolongation operator; and a change of the data structure of the solution found on exit. The zebra relaxation optimized in this way still is the most costly part of a multigrid cycle on the Cray.

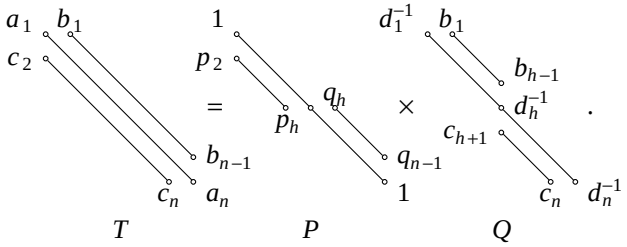
### 5.3: Parallel computers: twisted zebra relaxation

As mentioned in the previous subsection we can perform all operations simultaneously, thus vectorized, when solving multiple uncoupled tridiagonal systems. At the cost of decreased vector lengths and consequently a decreased vector-performance, we can in addition distribute the multiple uncoupled tridiagonal systems over multiple (vector-)processors. Before distributing the uncoupled tridiagonal systems we first apply a technique, which allows us to distribute the work of a (set of uncoupled) tridiagonal system(s) over two processors at no extra cost. Both the scalar operation count and the vector lengths remain the same. If we have more than two processors we simply divide the number of tridiagonal systems (the vector lengths) by half the number of processors. This way we can keep all (vector-)processors busy. For vector-processors the vector-lengths are now reduced by a factor equal to half the number of processors instead of by the total number of processors.

Consider the tridiagonal system:

$$Tx = y.$$

Instead of the standard $LU$-factorization, the matrix $T$ may also be factored as $T = PQ$, starting the factorization simultaneously from the top and the bottom of $T$, in the following way:




As we will show, the decomposition of $T$ as well as the back substitutions, can be carried out almost entirely in parallel using two processors. Note, that this parallelism is introduced without introducing additional work. A remarkable aspect is that this technique often leads to slightly more accurate solutions than the standard process

does. This technique has been proposed, in slightly different forms, in [1] for the accurate solution in 1 point and in [8] for parallelism. An analysis, which includes stability aspects is given in [16].
Defining $h := \lfloor n/2 \rfloor + 1$, we can effectively compute the *PQ* decomposition as follows:

| **Processor 1** $i = 2\ (1)\ h-1$ | **Processor 2** $j = n-1\ (-1)\ h+1$ |
|---|---|
| $d_1 = \frac{1}{a_1}$ | $d_n = \frac{1}{a_n}$ |
| $p_i = c_i d_{i-1}$ | $q_j = b_j d_{j+1}$ |
| $d_i = \frac{1}{a_i - p_i b_{i-1}}$ } $i$ | $d_j = \frac{1}{a_j - q_j c_{j+1}}$ } $j$ |
| $p_h = c_h d_{h-1}$ | $q_h = b_h d_{h+1}$ |
| $d_h = \frac{1}{a_h - p_h b_{h-1} - q_h c_{h+1}}$ | |

Note, that Processor 1 starts from the top of the matrix *T*, just like a regular *LU*-decomposition. The double horizontal bar depicts a *barrier* or *synchronization point* for the concurrency. Before being able to compute $d_h$ (on a single processor) both Processor 1 and 2 have to be finished. Storing the reciprocal of the diagonal elements during the decomposition stage, we avoid an expensive (vector-)division during the 'outward' substitution.

The tridiagonal system is solved by an 'inward' substitution $Pz = y$:

| **Processor 1** $i = 2\ (1)\ h-1$ | **Processor 2** $j = n-1\ (-1)\ h+1$ |
|---|---|
| $z_1 = y_1$ | $z_n = y_n$ |
| $z_i = y_i - p_i z_{i-1}$ } $i$ | $z_j = y_j - q_j z_{j+1}$ } $j$ |
| $z_h = y_h - p_h z_{h-1} - q_h z_{h+1}$ | |

followed by an 'outward' substitution $Qx = z$:

| $x_h = z_h d_h$ | |
|---|---|
| **Processor 1** $i = h-1\ (-1)\ 1$ | **Processor 2** $j = h+1\ (1)\ n$ |
| $x_i = (z_i - x_{i+1} b_i) d_i$ } $i$ | $x_j = (z_j - x_{j-1} c_j) d_j$ } $j$ |

The computational complexity of a twisted *PQ*-decomposition followed by an inward substitution and an outward substitution is the same as for the *LU*-decomposition followed by a forward and a backward substitution.

### 5.4: Parallel vector computers: twisted even/odd y-zebra relaxation

Combining the algorithmic choices described in the previous three subsections, we end up with a twisted-even/odd-y-zebra-relaxation for a parallel vector computer. Even/odd zebra yields a minimal scalar operation count for a resulting multigrid cycle because of the multigrid context. Y-zebra gives us the optimal vector-performance. Twisted tridiagonal system solving gives us parallelism for two processors at no extra cost. Using more than two processors decreases the vector-performance per processor, but this is more than counter-balanced by the multi-processor speedup.

## 6: Automatic vectorizing and parallelizing in standard Fortran

In order to optimize a program on a parallel vector-computer the ultimate goal we want to reach is a fully parallelizable, fully vectorizable program with a minimal scalar operation count. Ideally we would even choose the same algorithms on a scalar uni-processor computer, although we would use different optimization techniques cf. [14]. Designing such an algorithm we have to keep all three restrictions in mind. Clearly we are often working with conflicting requirements. Typically on parallel vector-computers, the most significant performance gain can be reached using vectorization. Implementing a parallelizable, vectorizable program we first try to minimize the scalar operation count while reaching maximal vectorization. Once we reached maximal vectorization, we try to obtain maximal parallelization again with a minimal introduction of additional work.

### 6.1: Minimizing the scalar operation count

In Section 5.1 we described how the choice of even/odd zebra relaxation reduced the scalar operation count of the restriction operator. We already used this technique for the original version of MGZEB, but it is a perfect illustration in this subsection. In [12] we separately describe the construction of a new algorithm for computing Galerkin coarse-grid approximations using a minimal scalar operation count. Furthermore, we were also able to decrease the scalar operation count of the data structure reordering. Finally, we became aware that we used an unneeded vector operation while computing the original line-*LU*-decompositions, so we omitted this during the twisted line-*PQ*-decompositions. After careful reinspection of every routine we are now convinced, that every single routine we use has a minimal scalar operation count. On a scalar uni-processor we would make the same algorithmic choices.

### 6.2: Vector tuning on a vector-register machine

For a general introduction on vectorizing programs we refer to [9]. Although the original version of MGZEB was written in fully autovectorizable Fortran it was tuned on the Cyber 205, one of the few memory-to-memory vector-computers. In this subsection we will focus on tuning existing vectorizable programs on vector-register machines.

**6.2.1: Optimal chaining:** The Cyber 205 has large vector-instruction-startup times and has virtual memory. In order to minimize the number of vector instruction startups, and possibly reducing the number of page faults we could implement the residual computation $f - Au$, using 7 loops, one for each diagonal, at no extra cost. Using the same implementation on a vector-register machine we have to perform 3 loads, 2 FLOPs and 1 store per diagonal. On a Cray X-MP or Cray Y-MP, this implementation would take 14 chimes because of the 3 loads per diagonal. Implementing it straightforwardly, the intermediate result can be kept in a vector-register (saving vector-register-stores and vector-register-loads) and we can optimally chain vector-operations. Implemented in this way, the same computation takes 7 chimes. The actual improvement over the original Cyber 205 implementation was less, because we handled three groups of diagonals simultaneously, which would take 9 chimes on the Cray. Although the actual timings and possible instruction overlaps might differ, the same holds for similar constructs on vector-register machines in general.

**6.2.2: Explicit loop collapsing:** The programmer can exploit explicit loop collapsing by initializing unused matrix elements to zeroes or by adding dummy grid-lines to make dimensions of arrays compatible. This technique is called *padding*. For the following example assume a vector-register-length of 64. If only the inner loop of the following loop nest is vectorized, we would need 130 vector-register-stores:

```
      REAL A(65,65)
        . . .
      DO 20 J = 1, 65
        DO 10 I = 1, 65
          A(I,J) = 0.
10      CONTINUE
20    CONTINUE
```

whereas the explicitly collapsed version

```
      DO 10 I = 1, 65*65
        A(I,1) = 0.
10    CONTINUE
```

would only need 67 vector-register-stores.
Clearly $65*65$ is a worst case example, but it can actually occur while performing multigrid iteration.

**6.2.3: Avoiding explicit vector temporaries:** On a vector-register machine it is often worthwhile to use scalar temporaries and to rely on automatic vector promotion by the compiler. Consider the following implementation of a vector swap:

```
      DO 10 I = 1, N
        T(I) = B(I)
        B(I) = A(I)
        A(I) = T(I)
10    CONTINUE
```

as opposed to

```
      DO 10 I = 1, N
        T    = B(I)
        B(I) = A(I)
        A(I) = T
10    CONTINUE
```

In the first version of the vector swap the vector temporary has to be stored in memory (assuming T is a non-local scratch array). In the second version the compiler only has to save the last value B(N) in T. If the local scalar T is only referenced in this loop, an optimizing compiler might even omit saving of this last value. On a Cray Y-MP the first version needs 3 chimes, because we have to perform 3 stores, whereas the second version only needs 2 chimes, because we only have to perform 2 (vector-) stores. For the MFLOP-rate-lovers: both variants perform at 0 MFLOP/s, so MFLOP-rates do not say everything.

**6.2.4: Vectorization directives:** This is not typical to tuning on a vector-register machine, but we will mention it here. Because we removed all aliasing in the final code, as will be explained later, we explicitly had to add CDIR$ IVDEP to allow vectorization in the data structure reordering. Using CF77 without invoking FPP, CFT77 sometimes used a computed safe vector length. This means, that the code is conditionally vectorized. At these points, we also manually inserted CDIR$ IVDEP directives.

## 6.3: Parallelizing on the Cray Y-MP in standard Fortran

On the Cray series of computers a programmer can make use of three different parallelization techniques. The oldest technique, *macrotasking*, requires programmers to modify their codes to exploit parallelism by doing extensive data scoping and requires the insertion of special library calls. The second oldest technique, *microtasking*, requires less data scoping and compiler directives replace the special library calls. An advantage of microtasking is that it requires programmers to change working programs much less. The most recent technique, *autotasking*, can be fully automatic; that is, it does not require programmer intervention, although the programmer is free to interact with the autotasking system by manually inserting compiler directives. Autotasking can exploit parallelism on the loop level without extending to subroutine boundaries, as microtasking is written to.

On Cray Y-MP systems the approximate number of clock periods to invoke and terminate parallel regions is 800. Consequently, the total number of clock periods must be at least 800 before adding one more processor will produce a speedup.

Since our goal is designing an autoparallelizable autovectorizable autonomous multigrid code, we use autotasking on the Cray. Except for a few explicitly inserted

directives, which most probably differ on other machines, the final code will autoparallelize on other parallel shared memory (vector-) computers as well. On an Alliant FX/8 or FX/80 for instance, where parallel overhead is almost neglectable, the final code also autoparallelizes.

**6.3.1: Removing Cyber 205 relics:** As mentioned before, our starting point was the already autovectorizable MGZEB code tuned on the Cyber 205. We had to remove two constructs specially devised for the Cyber 205.

- The Cyber 205 'possibly recursive syndrome'
  FTN200, the Cyber 205 Fortran compiler, suspected many loops of being possibly recursive, e.g.:

```
      REAL A(65,65)
        . . .
      DO 20 J = 2, 65
        DO 10 I = 1, 65
          A(I,J) = A(I,J) + A(I,J-1)
10      CONTINUE
20    CONTINUE
```

  We willingly violated the ANSI Fortran 77 standard (and even properly documented it) and used aliasing to hide the possible dependency for the FTN200 compiler.

```
      CALL FOO(A,A)
        . . .
      SUBROUTINE FOO(A, ALSOA)
        . . .
      DO 20 J = 2, 65
        DO 10 I = 1, 65
          A(I,J) = A(I,J) + ALSOA(I,J-1)
10      CONTINUE
20    CONTINUE
```

  Of course this worked on every machine we tried.

  The first time we ran this program, using FX/Fortran on an Alliant FX/4, and later, using the autotasking Cray compiler, we were painfully reminded, why the Fortran standard prohibits aliasing: allowing compiler optimization. Formulated the second way, the inner loop is properly vectorized, but the outer loop does not show its actual recursion and is unwantedly parallelized. We were deservedly punished for trying to outsmart one particular compiler.
- Cyber 205 longloop stripmining.
  On the Cyber 205 a 'long' loop, i.e. a loop with an iteration count larger than 65535 had to be stripmined explicitly. Because VAST-2, Pacific Sierra's vectorizing Fortran pre-compiler, was not at our disposal, we used the De Zeeuw-stripmining:

```
      DO 10 K = KB, KE
        . . .
```

  can be replaced (mechanically) by

```
      DO 10 KK = KB, KE, 65535
      KKE = (KK-1) + MIN(65535, KE - (KK-1))
      DO 10 K = KK, KKE
        . . .
```

  The Alliant FX/Fortran compiler default uses co-vi optimization: Concurrent Outer - Vector Inner. The autotasking Cray compiler uses the same optimization scheme. This means that an in this way stripmined vectorizable loop with an iteration count of $257 \times 257$ performs the first 65535 vector-iterations on one processor and the remaining 514 vector-iterations on a second processor, leaving every possible other processor idle. This is far from optimal but is easily fixed by removing the stripmining.

**6.3.2: General optimization techniques:** We used three techniques, also applicable for vectorization and frequently used by vectorizing compilers.

- loop peeling and loop fusion
  As mentioned before, on the Cray the overhead of parallel regions is significant, so apart from vectorizing inner loops one also has to minimize the number of outer loops, to a certain extent, while maintaining parallelizable outer loops. Since we are working on a two dimensional grid we often have multiple successive loop nests running over the interior region possibly including one or more boundaries and/or corners. We can also handle the boundaries and corners separately, this technique is called *loop peeling*, so only loops over the interior region remain. Assuming the remaining loops are not data dependent and run over the same dimensions, we can use one loop nest with the collected original loop bodies. This technique is called *loop fusion*. One more place where we used this in our code is on operations using the explicitly changed data structure. The data structure we are using discriminates two kinds of grid-lines: even and odd numbered vertical grid-lines. To simplify our implementation we assumed an odd number of vertical grid-lines. By first peeling the right boundary, we sometimes fused the loops over the even and odd numbered vertical grid-lines.
- loop splitting
  Everybody is aware that when vectorizing the following loop explicitly:

```
      DO 10 I = 2, 65
        A(I) = A(I) + A(I-1)
10    CONTINUE
```

  wrong results will be produced, because old values of A(I–1) recur.

Analogously, the following loop:

```
      DO 10 I = 1, 64
        A(I) = A(I) + A(I+1)
10    CONTINUE
```

might yield incorrect results when explicitly parallelized asynchronously, because we might use new values of A(I+1). In our code we encountered this problem while implementing the autoparallelizing Galerkin approximation described separately in [12]. Here all inner loops were vectorized, but two of the four loop nests could concurrently update matrix elements on adjacent horizontal lines. In order to avoid this situation, without requiring synchronized parallel loops, we simple applied explicit *loop splitting.*

**6.3.3: Autotasking directives:** After applying all techniques mentioned in the previous subsections, most loop nests automatically perform inner-loops vectorized and outer-loops parallelized. At four instances we had to manually insert autotasking directives in order to force parallelization on the Cray.

- inner loop autotasking
  The longloop stripmining mentioned in Section 6.3.1 was almost exclusively used for explicitly collapsed loops mentioned in Section 6.2.2. On the Cray, the compiler quite rightly only autotasks non-inner loops by default. All loops we previously stripmined on the Cyber 205 are perfect candidates for inner loop autotasking. There are two means of achieving this. The first one is using a global compiler switch: 'cf77 –Zp –Wd–ei', but this generates conditional code for every single not otherwise autotasked inner loop. We settled for using the compiler directive: CFPP$ INNER. The advantage of autotasking explicitly collapsed vector loops is that it is much easier to obtain a perfect load-balance, than for the original loop nest.
- parallel case
  For the actual parallel implementation of the twisted decompositions and the inward and outward backsubstitution we explicitly used CMIC$ CASE / CMIC$ END CASE in a parallel region. Explicit data scoping of one of the arrays was also necessary, because autoscoping gave the warning that it was using a PRIVATE array, which naturally yields correct results but implies copying a whole array which is unnecessary in this case.
- concurrent call
  Apart from twisting we also wanted to distribute the twisted tridiagonal systems over multiple processors. Because autotasking does not allow us to directly nest parallelism by using directives only, we were forced to create a separate routine for the twisted tridiagonal systems solving on two processors, and we had to use a CFPP$ CNCALL directive to allow concurrent calling. This sounds perhaps worse than it is: autotasking would automatically generate subroutines; it increases the readability of the original code; and much work is performed by one call of this routine. In fact most time is spent in this routine.
- possible data dependencies
  Because we removed all aliasing in the final code, we explicitly had to add CFPP$ NODEPCHK directives to allow autotasking in the *in-place* data structure reordering of the matrix *A*.

## 7: The performance of autotasked MGZEB on a Cray Y-MP4/464

In the following experiments we solve the Poisson equation on the unit square with Dirichlet boundary conditions and the right hand side constructed according to the exact solution $x(1-x)+y(1-y)$. The boundary conditions are eliminated. By setting $p=0$, $\sigma=1$ and $q=1$ (the recommended values that can also be found in [4, 18]) we obtain the so-called sawtooth multigrid cycle. We use a zero initial estimate and use a $3\times 3$ coarsest grid. If not stated otherwise, we perform multigrid cycling until the residual $l_2$ -norm becomes smaller than $10^{-10}$.

**Definition** *T*ime.
We define $T_{mode}$ to be the time used in *mode*-mode, where *mode* is one of $\{scalar, vector, parallel\}$. In *scalar*- and *vector*-mode we measure CPU time using 1 processor. In *parallel*-mode we measure wall-clock time using a dedicated machine. Using 1 processor of a dedicated machine the CPU time is equal to the wall clock time.

**Definition** *vector-S*peedup.

$$S_{vector} = \frac{T_{scalar}}{T_{vector}}$$

This is a measure for the vectorizability.

**Definition** *parallel-S*peedup.

$$S_p = \frac{T_{vector}(\text{using 1 processor})}{T_{parallel}(\text{using } p \text{ processors})}$$

Here we use $T_{vector}$ instead of $T_{parallel}$ on 1 processor to obtain a fair measure for the parallel speedup. Since parallel overhead is not always negligible, we sometimes measure $S_1 < 1$. Ideally $S_p = p$: this is what we call linear or perfect speedup.

For our experiments we used a locally available Cray Y-MP4/464 (sn1521) running UNICOS 6.0.12. The CF77 compiling system used: FPP Version 5.0, FMP Version 5.0.1 and CFT77 Version 5.0.0.12. *Scalar* results on a single processor are produced using 'cf77 –Wf–onovector'. *Vector* results on a single processor are produced using 'cf77 –Zv'. *Parallel* results are produced using 'cf77 –Zp', while varying the NCPUS environment variable.

### 7.1: The effect of minimizing the scalar operation count and vector-tuning on a Cray Y-MP

In the following table we show the speedup of the original MGZEB code (*old*) and the vector performance of the new version (*new*). We are solving a $257 \times 257$ finest grid problem. The vector lengths range from $257^2$ for collapsed loop nests on the finest grid to $\lfloor 3/2 \rfloor$ for inner loops on the coarsest grid. However, most of the computational work (75%) is performed on the finest grid (the computational complexity is linear in the number of grid points). The rows called 'preparation' and 'multigrid iteration cycles' are subtotals of the directly following indented routines.

| routine | $\frac{T^{old}_{vector}}{T^{new}_{vector}}$ | $S^{new}_{vector}$ | $T^{new}_{vector}$ (msec) |
|---|---|---|---|
| preparation | 2.47 | 10.89 | 25.779 |
| data structure change | 1.64 | 7.45 | 12.126 |
| Galerkin | 3.75 | 14.30 | 10.186 |
| decompositions | 1.53 | 13.10 | 3.424 |
| multigrid iteration cycles | 1.27 | 15.67 | 214.339 |
| zebra relaxation | 1.19 | 17.52 | 125.970 |
| residual | 1.42 | 17.82 | 38.486 |
| prolongation | 1.58 | 12.04 | 25.257 |
| restriction | 1.14 | 7.34 | 16.203 |
| norm | 1.00 | 8.21 | 4.789 |
| TOTAL | 1.40 | 15.16 | 240.118 |

The most prominent speedup is achieved in 'Galerkin' (cf. [12]) where the scalar operation count ratio $W_{old}/W_{new} > 170/74 \approx 2.3$. For the other two preparational routines the most significant part of the speedup was also reached by minimizing the scalar operation count. All other speedups were obtained by the vector-register tuning of the code. The 'multigrid iteration cycles' take 89% of the total time. Here we find $T_{old}/T_{new} \approx 1.27$; this improvement is achieved by vector-register tuning only. It is clear we could not improve on 'norm', a simple SDOT-operation.

### 7.2: The effect of autotasking

In the following table we show the parallel speedup of the new autotasked MGZEB code. We are solving a $513 \times 513$ finest grid problem. As in the previous table, the rows called 'preparation' and 'multigrid iteration cycles' are subtotals of the directly following indented routines.

| routine | $T_{vector}$ (msec) | $S_1$ | $S_2$ | $S_3$ | $S_4$ |
|---|---|---|---|---|---|
| prepar | 93.173 | 1.00 | 1.78 | 2.29 | 2.85 |
| change | 43.639 | 1.00 | 1.60 | 2.00 | 2.27 |
| Galerk | 37.057 | 1.00 | 1.97 | 2.89 | 3.72 |
| decomp | 12.434 | 1.01 | 1.98 | 2.08 | 3.58 |
| cycles | 734.359 | 0.99 | 1.82 | 2.55 | 3.09 |
| zebra | 437.566 | 0.99 | 1.77 | 2.46 | 2.92 |
| residu | 145.376 | 1.00 | 1.94 | 2.85 | 3.59 |
| prolon | 82.039 | 0.99 | 1.91 | 2.75 | 3.54 |
| restri | 46.459 | 0.97 | 1.76 | 2.44 | 3.00 |
| norm | 17.783 | 0.94 | 1.86 | 2.75 | 3.61 |
| TOTAL | 827.532 | 0.99 | 1.81 | 2.52 | 3.06 |

As described in the introduction of Section 7 we find $S_1 \leq 1$ (considering 1.01 to be a measuring fault).

The 'data structure change' is 100% vectorized. However, because the data structure change is done in-place (avoiding copying and the necessary workspace) some outer loops contain real data dependencies and have to perform sequentially. Hence the lowest parallel speedup $S_4 = 2.27$. All other routines either perform outer loops in parallel or are explicitly parallelized by distributing the work over multiple processors ('decompositions' and 'zebra relaxation').

For the 'Galerkin' coarse-grid approximations we already measured a perfect speedup factor of 4 once, so we suspect some problems with the 'dedicatedness' of the dedicated Cray we are using. At present we are working on this problem with people from SARA and from Cray.

The 'decompositions' are implemented using 4 parallel cases (2 for the twist $\times$ 2 for the split in the data structure), which explains $S_2 = S_3 \approx 2$. Given the coarse grain parallelism in 'decompositions', $S_2 = 1.98 \approx 2$ and $S_4 = 3.58 < 4$ suggest some 'dedicated' problems.

For the experimental results of the 'zebra relaxation' presented in the table above we used a hybrid strategy for the parallelism. On 2 processors we solve all tridiagonal systems exploiting the twisting for the parallelism. Since we can only achieve an $S_3 \leq 2$ by exploiting the twisting only, we distribute the multiple tridiagonal systems over three processors without exploiting the twisting for parallelism, hence we reduce the vector lengths by a factor 3. On 4 processors we first implemented it as described in Section 5.3 and 5.4. However, actual experiments indicate that *at present* distributing the multiple tridiagonal systems over 4 processors without exploiting the twisting for parallelism gives significantly better results. We emphasize *at present* because, as already stated above, we are suspecting some problems with the absolute dedicatedness of the Cray. The zebra relaxation consists of two parts: the tridiagonal systems right-hand-side computation is analogous to the 'residual' computation ($S_4 = 3.59$); the distributed/twisted tridiagonal system solution has a large granularity analogous to the 'decompositions' ($S_4 = 3.58$ and even $S_2 = 1.98$!); so it seems reasonable to expect $S_4 \approx 3.5$ at least. We actually measured $S_4 = 2.92$ and

since the zebra relaxation takes 53% of the total CPU time this has significant effects on the overall parallel speedup.

Although we still expect more, the overall parallel speedup $S_4 > 3$ is quite satisfactory, but we expect to improve on this in the very near future.

## 8: Conclusions

We were able to improve a vectorizable algorithm for the Galerkin coarse-grid approximation by introducing a new optimally efficient algorithm (scalar operation count, vectorization and parallelization).

100% Vectorizable code does not necessarily mean the code is optimal on a given vector-machine. We were able to improve the vectorization of the already 100% vectorizable original version of MGZEB (achieving a speedup of 1.3 for the 'multigrid iteration cycling' cf. Section 7.1) by applying some guidelines described in Section 6.2.

The original MGZEB code was autovectorizable and still is. Now it is also autoparallelizable, except that at two points we had to explicitly add compiler specific directives to perform loops in parallel. These places are also the only places where we directly make use of the number of processors.

### Acknowledgements

Financial support for this work was provided by Cray Research, Inc. under a 1991 University Research & Development Grant for the project 'Numerical Multigrid Software for the Cray Y-MP4'. This work was sponsored by the Stichting Nationale Computerfaciliteiten (National Computing Facilities Foundation, NCF) for the use of supercomputer facilities, with financial support from the Nederlandse Organisatie voor Wetenschappelijk Onderzoek (Netherlands Organization for Scientific Research, NWO). We are grateful to the staff of SARA (the Academic Computer Centre Amsterdam), for providing us with dedicated Cray time. We wish particularly to acknowledge the dedicated help of Bert van Corler and Jan Overweel of SARA. Last but not least, we wish to thank Herman te Riele for careful reading and commenting several draft versions of this paper.

### References

1. I. Babuška, ''Numerical Stability in Problems of Linear Algebra,'' *SIAM J. Numer. Anal.* **9**(1) pp. 53–77 (1972).
2. W. Hackbusch, *Multi-Grid Methods and Applications*, Springer-Verlag (1985). Springer Series in Computational Mathematics 4.
3. P.W. Hemker, ''The incomplete LU-decomposition as a relaxation method in multigrid algorithms,'' pp. 306–311 in *Boundary and interior layers - Computational and asymptotic methods*, ed. J.J.H. Miller, Boole Press (1980).
4. P.W. Hemker, ''On the comparison of line-Gauss-Seidel and ILU-relaxation in multigrid algorithms,'' pp. 269–277 in *Computational and asymptotic methods for boundary and interior layers*, ed. J.J.H. Miller, Boole Press (1982).
5. P.W. Hemker, R. Kettler, P. Wesseling, and P.M. de Zeeuw, ''Multigrid methods: development of fast solvers,'' *Appl. Math. Comp.* **13** pp. 311–326 (1983).
6. P.W. Hemker, P. Wesseling, and P.M. de Zeeuw, ''A portable vector-code for autonomous multigrid modules,'' pp. 29–40 in *PDE SOFTWARE: Modules, Interfaces and Systems*, ed. B. Engquist & T. Smedsaas, North-Holland (1984).
7. P.W. Hemker and P.M. de Zeeuw, ''Some Implementations of Multigrid Linear System Solvers,'' pp. 85–116 in *Multigrid methods for integral and differential equations*, ed. D.J. Paddon and H. Holstein, Oxford University Press (1985).
8. G.R. Joubert and E. Cloete, ''The Solution of Tridiagonal Linear Systems with an MIMD Parallel Computer,'' *ZAMM Z. Angew. Math. u. Mech.* **65**(4) pp. T383–T385 (1985).
9. J.M. Levesque and J.W. Williamson, *A Guidebook to Fortran on Supercomputers*, Academic Press (1989).
10. W.M. Lioen, ''NUMVEC FORTRAN Library manual, Chapter: Elliptic PDEs, Routine: MGZEB,'' NM-R8518, CWI (1985).
11. W.M. Lioen, ''Multigrid methods for elliptic PDEs,'' pp. 181–198 in *Algorithms and Applications on Vector and Parallel Computers*, ed. H.J.J. te Riele, Th.J. Dekker, and H.A. van der Vorst, North-Holland (1987).
12. W.M. Lioen, *An Optimally Efficient Algorithm for Galerkin Coarse-Grid Approximation*, CWI (to appear).
13. M. Louter-Nool, *MGD1M, a Parallel Multigrid Code with a Fast Vectorized ILU-Relaxation*, CWI (to appear).
14. M. Metcalf, *FORTRAN Optimization*, Academic Press (1982). APIC studies in data processing 17.
15. K. Stueben and U. Trottenberg, ''Multigrid methods: fundamental algorithms, model problem analysis and applications,'' pp. 1–176 in *Multigrid methods. Procs. Koeln-Porz, 1981*, ed. W. Hackbusch & U. Trottenberg, Springer-Verlag (1982). Lect. Notes in Math. 960.
16. H.A. van der Vorst, ''Analysis of a parallel solution method for tridiagonal linear systems,'' *Parallel Comput.* **5**(3) pp. 303–311 (1987).
17. P. Wesseling, ''Theoretical and practical aspects of a multigrid method,'' *SIAM J. Sci. Stat. Comp.* **3**(4) pp. 387–407 (1982).
18. P. Wesseling, ''A robust and efficient multigrid method,'' pp. 614–630 in *Multigrid methods. Procs. Koeln-Porz, 1981*, ed. W. Hackbusch & U. Trottenberg, Springer-Verlag (1982). Lect. Notes in Math. 960.
19. P. Wesseling, *An introduction to multigrid methods*, John Wiley & Sons (1992).
20. P.M. de Zeeuw, ''NUMVEC FORTRAN Library manual, Chapter: Elliptic PDEs, Routine: MGD1V and MGD5V,'' NM-R8624, CWI (1986).