APPLIED MATHEMATICS AND COMPUTATION 13:215-227(1983)

0096-3003/83/$03.00

VECTORIZED MULTIGRID POISSON SOLVER FOR THE CDC CYBER 205*

D. Barkai**, A. Brandt***
**Control Data Corporation, Institute for Computational Studies at Colorado State University, PO Box 1852, Fort Collins, Colorado 80522; ***Weizmann Institute, Department of Applied Mathematics, Rehovot, Israel 76100.

ABSTRACT

The full multigrid (FMG) method is applied to the two dimensional Poisson equation with Dirichlet boundary conditions. This has been chosen as a relatively simple test case for examining the efficiency of fully vectorizing of the multigrid method. Data structure and programming considerations and techniques are discussed, accompanied by performance details.

April 1983

1. INTRODUCTION

The multigrid (MG) method has been shown to be a very efficient solver for discretized PDE boundary-valve problems on serial (scalar) computers. However, it was not clear how well can the MG approach be adapted to execute effectively and efficiently on a vector processor, such as the CDC CYBER 205, where considerations other than operations-count may play an important role. The purpose of this paper is to report our experience in implementing an MG code on the CDC CYBER 205. More specifically, the test-case considered is the two-dimensional Poisson equation with Dirichlet boundary conditions. It will be assumed here that the reader has some familiarity with the philosophy, the motivation and the basic computational processes of MG as a fast solver. These processes are described in detail in a number of papers in these proceedings and [1] and [2] and references therein. The algorithm described in this paper is basically the same as the one given in the appendix of [3], whose description is detailed in sections 8.1 and 6.4 of [3]. Therefore, no full description of the MG algorithm is given here, but the relevant details are included in the appropriate context. The main emphasis of this paper is the vectorization of these processes. Thus, we will not assume an in-depth knowledge or experience in applying MG solvers on a vector-processor type of a computer system.

---

* Presented at the International Multigrid Conference, Copper Mountain, Colorado, April 6-8, 1983.

Consequently, Section 2 contains a brief summary of architectural and conceptual features of a vector processor (specific to the CDC CYBER 205), which are relevant to this application, as well as software tools available for a tight correlation between the hardware and the computational process. Sections 3, 4 and 5 are devoted to the description of the techniques used for vectorizing the procedures for the relaxation, the residual transfer calculation and the interpolation, respectively. The total full multigrid (FMG) process and various parameters and constraints are described in Section 6 interleaved with convergence and timings (performance) details. Finally, Section 7 contains some concluding remarks and comments regarding future plans.

2. VECTOR PROCESSING

The most significant difference between a traditional, serial computer and a vector processor is the ability of the latter to produce a whole array ("vector") of results upon issuing a single hardware instruction. The input to such a vector-instruction may be one or two vectors, one or two elements ("scalars"), or a combination of the above. The instructions fall into two main categories--those that perform floating-point arithmetic (including square root, sum, dot-product, etc., as well as the basic operations), and those which may be collectively called "data-motion" instructions. These may be used, for example, to "gather" elements from one array into another using an arbitrary "index-list"; to "compress" or "expand" an array; to "merge" two arrays into one (with arbitrary "interleaving" patterns), etc.

The need for vector data-motion instructions becomes apparent when one considers the definition of a vector on a CDC CYBER 205. A vector is a set (array) of elements occupying consecutive locations in memory. It means, by the way, that a vector may be represented in FORTRAN by a multi-dimensional array; i.e., a two- or three-dimensional array may be used in computations as a single vector. The reason for this vector definition is that when performing vector operations on the CDC CYBER 205 the input elements are streamed directly from memory to the vector pipes and the output is streamed directly back into memory without any intermediate registers.

The timing formula for completing a vector instruction contains two components. One is fixed, i.e., independent of the number of elements to be computed, and is called "start-up" time. In fact, it amounts to start-up and shut-down; it involves fetching the pointers to the input and output streams, aligning the arrays so as to eliminate bank conflicts and getting the first pair of operands to the functional unit (the pipe-line) and the last one back to memory. Typical time for the "start-up" component is 1 microsecond, or about 50 cycles (clock periods). The other component of the timing formula is the "stream-time" which is proporational to the number of elements in the vector. The result rate for a 2-pipe CDC CYBER 205 for an add or multiply is 2 results per cycle. It is apparent now that in order to offset the "wasted" cycles of start-up times it is beneficial to work with longer vectors. The system is better utilized if a single operation is performed on a long vector, rather than several operations to compute the same number of results. Given a vector length, N, one can evaluate the efficiency of the computation as the ratio between the number of cycles used to compute results and the total number of cycles the instruction has taken; i.e., $(N/2)/(N/2 + 50)$. The maximum vector length

the CDC CYBER 205 hardware allows is 65,535 elements. The start-up time becomes quite negligible long before that.

The vector "arguments" for vector instructions are inserted through a construct called Descriptor. It is a quantity occupying 64 bits which fully describes a vector through two integer values: one is the virtual address of the starting location of the vector, the other is the number of elements, or the length, of the vector. An element may be a bit, a byte, a half-word (32-bits) or a word (64-bits) depending on the intruction and the argument within the instruction. The CDC CYBER 205 FORTRAN provides the ability to declare variables of "type" Descriptor and Bit, as well as, extensions for assigning Descriptors to arrays and syntax for coding vector instructions without such an explicit association. Bit arrays occupy exactly one bit per element, since the CDC CYBER 205 is bit-addressable. Bit vectors are used for creating a "mapping" between an array containing numerical values and a subset of it. A Bit vector may be used to control a vector floating-point operation (hence the term "control-vector" which is commonly used for a Bit vector) as follows: Take, for example, an add operation. All the elements of the two input arrays are added up, but only those result elements where the corresponding element of the control-vector is 1 will be stored into the results vector. The other elements will not be modified. Alternatively, one may specify storing on zeros in the control-vector, and discarding results corresponding to a 1.

Another common use of bit vectors is associated with some of the data-motion instructions. Two examples will be given here: The "compress" instruction is used to create a vector which is a subset of another vector. This operation has two input descriptors--one points to a numeric vector, the other to a bit vector. Whenever a 1 is encountered in the bit-vector the corresponding numeric element is moved to the next location of the output vector, i.e., the input array is "compressed" (the reverse process may be accomplished with an "expand" instruction). A single bit-vector may also be used to "merge" two numeric vectors into one. The bit-vector is scanned and when a 1 is encountered the next element of the first input vector is put into the next location of the output vector, when a zero is found in the bit-vector the next element of the second input vector is moved into the next location of the output vector. The timing for both these instructions is dictated by the total length of the bit-vector. The result-rate is the same as that of vector arithmetic, i.e., on a two-pipe CYBER 205 it is two elemets per cycle (whether they are moved or not). It will be noted here that there are vector instructions for creating repeated bit patterns at a rate of 16 bits per cycle.

Before concluding this section let us briefly mention the existence of an "average" instruction, which computes an average of two vectors, or adjacent means of a single vector, at the rate of a single floating-point operation. One can also "link", for example, an add and a multiply operation, provided at least one of the three inputs is a "scalar", and perform the two operations as if it were only one. All the instructions mentioned above are directly available through Fortran in-line function calls.

3. RELAXATION

Now we are ready to examine the ways in which to utilize the tools and the vector processing concepts discussed in the previous section for vectorizing the Multigrid application. The success of such an exercise

hinges, to a large extent, upon the efficiency with which the relaxation process may be accomplished.

Discretization of the two-dimensational Poisson equation is achieved via the 5-points differencing scheme. Thus, assuming geometric interpretation of the indices for the moment, the set of the simultaneous equations to be solved may be written as

$$u_{i,j-1} + u_{i-1,j} + u_{i+1,j} + u_{i,j+1} - 4 * u_{i,j} = h^2 F_{i,j}$$

where u is the unknown function, h is the interval between two grid points (in either direction) and F is the right-hand side function. i varies from 2 to N_1-1 and j from 2 to N_2-1, where N_1 and N_2 are the number of grid points along the two directions.

One may want to consider the usual (lexicographic) Gauss-Seidel relaxation procedure. This, however, will be in conflict with vectorization, as may be easily deduced. The Gauss-Seidel relaxation is characterized by the use of updated values as soon as they become available. Vectorization means processing many such values in parallel, i.e., not waiting for the previous element to be updated. The obvious alternative is the red-black or checker-board ordering, where all the four neighbors of each point belong to the other "color". The convention used here is that the "color" of the grid points at the corners of the rectangle is red. The grid may accordingly be divided into two vectors and the relaxation performed in two stages: first, the values at red points are updated using "old" values, then the values at black points are updated using the "new" red values. Throughout the code the two vectors of the unknown function (and of the RHS function) are stored consecutively following each other, where inside each vector the values are stored column-wise as shown in Figure 1. This storage applies, of course, to all the grids used.

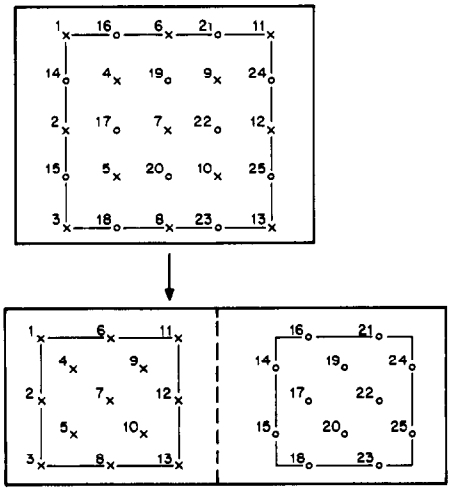

Figure 1. Mapping of the Lexicographic into the "Red-Black" Ordering. The dotted line indicates the separations of the grid points into two vectors.

The reader will notice that the vectors thus created are not confined to one column, but extend over the entire grid. It was done in order to achieve longer vectors in line with the desire expressed in Section 2. This, however, introduces the hazard of overwriting values residing on the boundary of the grid. To avoid this a bit control-vector was created for each grid, in a set-up routine, which contains zeros where boundary points exist and ones for interior points. We use this "boundary control vector" to assure storing new values only into the interior of the grid.

The computation requires the sum of the 4 neighbors for each grid point. One can easily verify that, using vector add operations this can be done with two operations only. One to add a vector into itself, with some offset (e.g., start with elements 2 and 5 in Figure 1) and the second to add the resultant vector into itself (with some other appropriate offset). The remaining calculation involves subtracting the result from the RHS values and multiply by a constant (being -0.25), which is accomplished as a linked-triad operation; the result is then stored into place under the control of the boundary bit-vector. Thus, each of the two stages (two "colors") requires three floating-point operations using vector length of, approximately, $(N_1 * N_2)/2$ elements long. In fact, some more savings in the computations occur in the first relaxation sweep after moving to a coarser grid, since the sum of the "neighbors" need not be computed for the first "color," being known to be zero. This is because we are beginning to compute a correction-function whose first approximation is zero. The vector-operations count for this relaxation sweep is thus reduced from 6 to 4. Also, when transferring a solution-function (not "correction") to a finer grid, as part of the FMG process, an interpolation can be used which will save the relaxation on the first "color" (see Sec. 5).

In conclusion, the relaxation process can obviously be done extremely fast on the CYBER 205. Timing details will be given in Section 6.

4. FINE TO COARSE RESIDUAL TRANSFER

Residuals have to be computed at those fine-grid points which also belong to the coarser grid. These residuals are directly transferred to the corresponding coarse-grid points weighted by 1/2 ("half injection"; the factor of 1/2 is motivated by the fact that the fine-grid residual is zero at black fine-grid points, hence the other residuals should be multiplied by 1/2 to represent the correct average). See Figure 2.

The computation involves four floating-point operations (two of them are linked triads) for evaluating the residuals of the red points on the finer-grid and multiplying them by 1/2. This, however, does not conclude the procedure. At this stage we need to apply the "compress" operation three times as follows: using a pre-defined bit-vector we extract the residual values corresponding to coarse-grid points, i.e, belonging to odd-numbered columns of the red section of the finer grid. (Note that we have thrown away half the calculated residuals. This procedure is both simpler and a little faster than having to perform all the compress operations needed for computing only the required residuals.) Now, as is evident from Figure 2, we have all the desired values for the coarser grid stored in lexicographic order. To separate them into "red" and "black" sections the "compress" instruction is applied twice (once for each color) using a pre-defined "picket fence" bit-vector. The procedure as described here produces optimum performance even though some redundant operations are

performed. The alternatives are to perform different (more "costly") data motions or to operate on much shorter vectors. Finally, another vector operation is executed to zero out the unknown function of the coarser grid in preparation for evaluating the correction function. In total the procedure requires 8 vector "start-ups" associated with 5 operations of approximate length of $(N_1 * N_2)/2$, and 3 operations of length $(N_1 * N_2)/4$, where N_1 and N_2 are the dimensions of the finer grid.

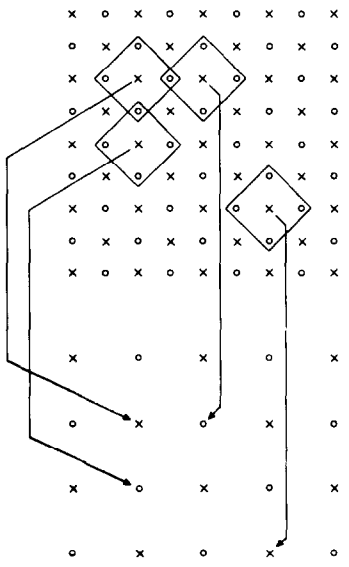

Figure 2. Transfer to a Coarser Grid: The residual calculation. Each "Box" contains the fine grid points involved in the computation for the corresponding coarse grid point.

5. INTERPOLATION

Interpolation, in the context of this paper, is the process by which we transfer from a given grid to a finer one. Two types of interpolations are employed here: Type I interpolation is used when a correction is interpolated from the coarser grid and added to the finer grid. The Type II interpolation is used to compute a first approximation on the finer grid, based on existing values on the coarser grid. The use of the red-black ordering, combined with the fact that a relaxation always follows an interpolation, implies that only one color of the finer-grid points need to be interpolated (the other color will be computed by a relaxation pass on that color).

Type I interpolation is bilinear employing points as shown in Figure 3. Only interior black points on the finer grid need to be evaluated. Due to the required averaging of the coarse grid values it is convenient to first merge the red and black points of this grid using the "picket-fence" bit vector to produce the lexicographic ordering. Next, two averages are

computed. The average over the coarse grid, where the two input vectors are offset by a column, will produce the quantities to be added into black points on even-numbered columns on the fine grid. A second average, where the offset between the two vectors is one element, is executed for fine grid black points corresponding to odd numbered columns. This last operation produces redundant values (at the end of each coarse grid column) which are thrown away using the "compress" operation with an appropriate pre-defined bit vector. The two resultant coarse grid "average-vectors" are then interleaved, using a "merge" instruction, under the control of the bit vector where the "1's" and "0's" correspond to odd and even columns, respectively. Finally, the merged values are added to "black" points of the finer grid under the control of the "boundary" bit-vector which inhibits storing values into the boundary of the grid. The whole procedure amounts to 3 floating-point operations, 2 "merges" and 1 "compress." The 6 vector operations may also be divided into 4 operations of length $(N_1 * N_2)/4$ and 2 operations of length $(N_1 * N_2)/2$, approximately. (N_1 and N_2 are the dimensions of the finer grid.)

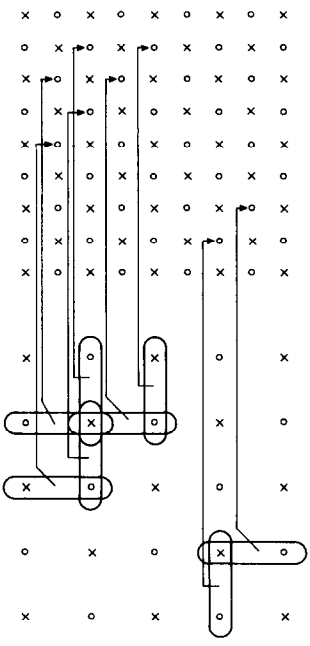

Figure 3. Type I Interpolation. It shows where averages of coarse grid values are added into "Black" points on the fine grid.

Type II interpolation is a 4th order one, described, for example, in section 6.4 of [3]. It produces new red unknown-function values on a finer grid using rotated difference operators. The values at the black points are produced by half a relaxation sweep, i.e., a relaxation pass over the fine-grid black points. (This pass may be regarded as part of the interpolation process. In the timing tables below, however, the time spent in this pass is counted as relaxation time.) The process is described pictorially in Figure 4. All the interior coarse grid values are moved to occupy

the corresponding fine-grid points. The relaxation operator is applied to these values in order to compute interior red points of the even-numbered columns on the fine grid. The only difference between the relaxation here and the one described in Section 3 is that the operator is the "rotated" 5-point Laplacian and the interval between each point and its neighbors is changed from h to $\sqrt{2}$*h. The RHS function values required for this relaxation are available from the fine grid RHS array (a "compress" operation is performed to retrieve even-numbered column values). The whole procedure, thus, requires 2 "merges" (one for merging red-black values of the coarse grid, the other for merging the "transferred" and "relaxed" values of the red fine grid points); 3 floating-point operations for the relaxation; 2 "compress" operations (one for throwing away redundant, incorrect averages and one for collecting RHS values); and, finally, one vector-move operation under the control of the boundary bit-vector for storing the new red fine grid values into place. Five out of the 8 vector operations have length of about $(N_1 * N_2)/4$, the other 3 are associated with a length of $(N_1 * N_2)/2$; N_1 and N_2 being the dimensions of the finer grid.

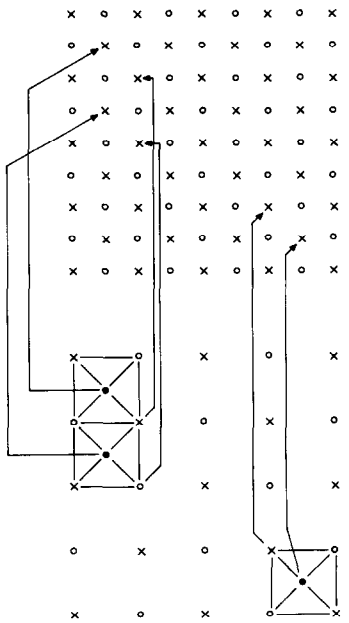

Figure 4. Type II Interpolation. Coarse grid values are transferred to odd numbered columns on the fine grid. These values are used to compute, via the relaxation operator, the even numbered column values.

6. PERFORMANCE AND CONVERGENCE

The basic computational procedures, studied in the previous three sections, can now be linked together to form the FMG process. Figure 5 is a schematic description of the sequence of events which leads to an approximate solution of the difference equations. The finest grid (where a solution is sought) is assigned the highest level number. The example

depicted in Figure 5 describes an FMG with 5 levels where the process starts at level number 2. This may not be necessary, as will be argued below, and one may visualize the FMG starting at a higher level simply by deleting the left-hand-side of the figure. This starting level is a parameter controlled by the user. The FMG shown in Figure 5 is composed of what is known as "V" cycles. In each "V" cycle one performs relaxation-residual calculation-relaxation...until reaching the coarsest grid, then a sequence of interpolation-relaxation is executed. The transfer from one "V" cycle to the next is achieved via Type II interpolation. More specifically, the FMG we implemented may be characterized as FMG (M,L,R1,R2,R3,R4), where M is the number of levels and L is the starting level; R1 and R2 indicate the number of relaxations before moving to a coarser grid and before moving to a finer grid, respectively. R3 and R4 have the same meaning and apply to the last "V" cycle only. All these parameters are provided by the user. The user may also specify the size of the coarsest grid to be used. It must have an even number of intervals in each direction. (In our experiments the coarsest grid had 3 by 3 points; i.e., 2 by 2 intervals.) The user also specifies the mesh size h (assumed to be the same in both directions) on the finest grid.

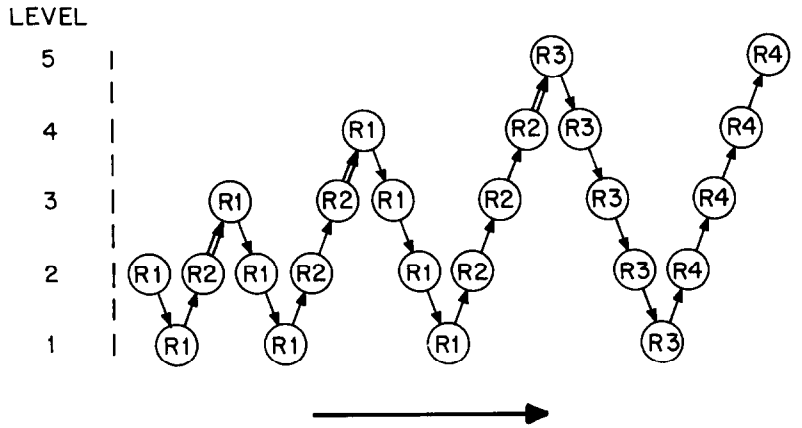

Figure 5. The Full Multigrid (FMG) Process: FMG (5, 2, R1, R2, R3, R4). The circles indicate the number of relaxations performed at a given level. Downwards arrow signifies residual calculation between relaxations, upwards arrow implies interpolation. (When a level is encountered for the first time the interpolation is of Type II, indicated by a double line above, otherwise it is of Type I.) When level 1 contains only one interior point only one relaxation sweep is performed thereon, regardless of the values given to R1 and R3.

The process described above is deterministic, in the sense that the user defines the steps to be taken, based on prior knowledge of the characteristics and smoothness of the function to be solved. It is also known that if L=2 the FMG guarantees a solution error smaller than the truncation error (introduced by the differencing scheme), for L_2 norm, for example. We have allowed, however, as a user-option, the evaluation of

the L_1, L_2 and L_∞ norms of the residual at various points. Testing was done for problems which have solution of the form:

$$C * \cos (k \ (x + 2y))$$

with and without the addition of a 6th degree polynomial which vanishes on the boundary. In all these cases the FMG process with L=2 indeed produced a solution with an algebraic error (error in solving the difference equations) much smaller than the truncation error, in the L_1, L_2 and L_∞ norms.

Only "V(2,1)" cycles were used for the results and timings to be quoted here. This turns out to be the optimum combination for the Poisson equation. More relaxations at each stage do not improve the final result enough to justify the additional work, less relaxations may cause deterioration in the accuracy. (If full weighting were used instead of half injection, the optimal cycle would be "V(1,1)". This would, however, be less efficient than the present procedure since full weighting is substantially more costly than a relaxation sweep.) In the performance details which follow, we will give results for various values of L since, in many cases, in particular when a reasonable initial guess is available, high values of L, even L=M, may provide sufficient accuracy. This is, in particular, the situation when the Poisson solver is used within some external iterative process, or at each time step of an evolution problem.

Before discussing the timings we should briefly mention some set-up procedures. A routine is provided for re-ordering the initial array (from lexicographic to red-black) if it is not so structured yet. This is done through two "picket-fence compress" operations and amounts to 0.185 msecs. for a 65 by 65 grid, for example. Putting the solution back into lexicographic order is done with a single "merge" instruction and takes half as long. Next, there is a routine which defines various pointers and lengths for all the grids used, as well as the bit-vectors discussed earlier. For many applications, where the solver is used many times with the same grid definition, this will be done only once. It will not, therefore, be included in the total times quoted below (it takes 0.29 msecs. for a 65 by 65 grid with 6 levels). The last set-up routine is included in the timings information. This routine defines the boundary values and the RHS for all the levels between L and M-1. It also sets the initial guess on the level L grid.

The code was run with grid sizes of 33 by 33, 65 by 65 and 129 by 129 (M = 5, 6 and 7, respectively) with L=2,...,M. Total execution times are given in Table 1. It shows, for example, that a 65 by 65 grid may be solved in as little as 1 msec., and, at most, in 2 msecs. By examining the processing time per grid-point one can see the effect of vector-instructions start-up times or the dependence of the performance upon vector lengths. On a serial processor the time per element would have been, approximately, a constant across each line in Table 1. We observe, however, that the processing of the 129 by 129 grid is roughly twice as efficient as that of the 33 by 33 grid. This is due to the fact that even though the number of vector "start-ups" remains nearly the same (across a given line), the number of elements solved for has increased by a factor of 16. Hence, more time is spent doing useful arithmetic in the vector pipelines.

TABLE 1. Execution times for various parameters of the FMG. The entries on the left are total times in milliseconds. The entries enclosed in parenthesis are the execution times in microseconds per grid-point (only interior points are taken into account).

M-L+1 (No. of "V"'s)	33 by 33 (M = 5)	65 by 65 (M = 6)	129 by 129 (M = 7)
1	0.360 (0.37)	1.006 (0.25)	3.293 (0.20)
2	0.604 (0.63)	1.552 (0.39)	4.910 (0.30)
3	0.729 (0.76)	1.810 (0.46)	5.440 (0.34)
4	0.801 (0.83)	1.947 (0.49)	5.687 (0.35)
5		2.009 (0.51)	5.807 (0.36)
6			5.875 (0.36)

Tables 2 and 3 present a more detailed analysis of timings for a single example, namely for solving a 129 by 129 grid with 7 levels and starting at level 2. The entries in Table 2 show timings in msecs. by level and by procedure. One notices that the total time spent performing relaxations is less than 50% of the total time. This is to be compared against the 80-90% of total time used for relaxations on a serial processor. This is, of course, due to the fact that the vectorized relaxation is extremely efficient and does not involve any data-motion operations. The interpolation and the residual calculations, though fully vectorized, involve some data-motion operations, and, therefore, consume a relatively higher proportion of the execution time than they would on a "scalar" computer. Another observation worth mentioning is that the contributions to all the procedures arising from levels 2 to 4 is roughly the same, even though the amount of work differs by a factor of 4 between levels. This is a consequence of the relatively short vectors which characterize the coarser grids. It also explains the larger weight the coarse grids have in the vectorized code compared to that of the serial process.

TABLE 2. Execution times in milliseconds for solving a 129 by 129 grid with starting level 2. Breakdown by procedure and by level. For the residual calculation and the interpolations the entry in the table corresponds to the finer grid involved.

Level	Grid Initialization	Relaxation	Residual Calculation	Interpolation Type I	Interpolation Type II	Total
1 (3x3)		0.010				0.010
2 (5x5)	0.011	0.179	0.014	0.011		0.215
3 (9x9)	0.015	0.160	0.060	0.049	0.024	0.308
4 (17x17)	0.034	0.189	0.068	0.053	0.028	0.372
5 (33x33)	0.106	0.320	0.117	0.095	0.053	0.691
6 (65x65)	0.388	0.690	0.261	0.194	0.141	1.674
7 (129x129)		1.257	0.497	0.357	0.494	2.605
TOTAL	0.554	2.805	1.017	0.759	0.740	5.875

In Table 3 we have measured the time in microseconds for each time a procedure is executed for a given level, accompanied by the number of times the procedure is performed. It should be noted here that when level 1 is involved in any of the procedures a scalar code was used, since it has only one interior point. Again, the effect of vector lengths is such that the level 3 relaxation is comparable to that of level 2, for example. Only when we get to the finest grids do we observe timing ratios which correspond to the ratios of the number of elements processed. The reader should be reminded that the average time of the relaxation procedure is not fully accurate, since some relaxations are not quite "complete" as was explained in Section 3 (i.e., after Type II interpolation and after residual calculation). The residual calculation takes longer than the relaxation (in contrast to the scalar case), which is understandable from the discussion in Sections 3 and 4.

TABLE 3. Procedure-calls count and average times in microseconds per call. Breakdown by levels for the 129 by 129 problem with starting level 2.

Note: Some of the relaxations are not "complete." (See Section 3)

	Relaxation		Residual		Interpolation Type I		Interpolation Type II	
Level	No.	Time	No.	Time	No.	Time	No.	Time
1 (3x3)	6	1.7						
2 (5x5)	18	9.9	6	2.3	6	1.8		
3 (9x9)	15	10.7	5	12.0	5	9.8	1	24.0
4 (17x17)	12	15.8	4	17.0	4	13.3	1	28.0
5 (33x33)	9	35.6	3	39.0	3	31.7	1	53.0
6 (65x65)	6	115.0	2	130.5	2	97.0	1	141.0
7 (129x129)	3	419.0	1	497.0	1	357.0	1	494.0

To conclude the performance discussion we will mention that the vectorized code executes about 15 times faster than the scalar version on the CDC CYBER 205, and roughly 500 times faster than the CDC CYBER 720.

The lesson from what was said above is that relaxations are relatively "cheap" in terms of execution times, and computations on the coarser grids are realtively "costly" (compared with the ratios found on scalar processors).

7. CONCLUDING REMARKS

One important lesson, known very well to those involved in vector processing, is that it demands careful data structuring and analysis of the "mapping" between the data and the operations to be performed, if the vector capabilities of the processor are to be efficiently utilized. We have also demonstrated that the traditional operations-count as a measure of processing time is not sufficient. On a vector processor one has to take into account the number of vector operations (or the lengths of the vectors) and the data-motion operations (which occur on a serial processor, too, but are often ignored when algorithms are evaluated). The result of the above is that one may have to re-examine the various parameters of the algorithm when migrating the Multigrid application from a serial to a vector processor. This aspect requires further investigation.

We feel that the experiment with the model-case studied in this paper was successful and the performance achieved very pleasing. It certainly warrants continuation work. Some obvious areas we intend to engage in are the following: Extending the application to three-dimensional Poisson equations; code a similar application to cater for the, more general, Diffusion equation; and implement "full-weighting" residual calculation and cubic interpolation. In addition one may, of course, generalize this work in many directions. More general boundary conditions (Neumann, etc.) can be implemented. The solution of non-linear problems (using FAS multigrid version) and systems of equations can also be vectorized in a similar fashion. More difficult, but potentially important, is the extension to general domains, which will require a lot of thought about data structures and data motion. As a last comment, it will be noted that all the timings quoted here were achieved using 64-bit arithmetic. On the CDC CYBER 205 one can use 32-bit arithmetic as well, and, thus, double the result rate for vector operations while halving the memory requirements. For the purpose of obtaining albebraic errors smaller than truncation errors in solving second order equations, the 32-bit arithmetic is indeed enough. We intend to examine this option.

REFERENCES

[1.] A. Brandt, "Multi-level adaptive solutions to boundary-value problems", Math. Comp. 31, (1977), 333-390.

[2.] W. Hackbusch and U. Trottenberg, ed., "Multigrid Methods", Proceedings of a Conference (Koln-Porz, Nov. 1981), Springer-Verlag, 1982.

[3.] K. Stuben, K. Trottenberg, "Multigrid Methods: Fundamental algorithms, model problem analysis and applications". In [2] pp. 1-176.