Implementation of an Electronic Structure Program System on the CYBER 205

Reinhart Ahlrichs, Hans-Joachim Böhm, Claus Ehrhardt, Peter Scharf, and Heinz Schiffer
Lehrstuhl f. Theor. Chemie, Universität Karlsruhe, D-7500 Karlsruhe, Federal Republic of Germany

Hans Lischka
Institut f. Theor. Chemie, Universität Wien, A-1090 Wien, Austria

Michael Schindler
Lehrstuhl f. Theor. Chemie, Ruhr-Universität Bochum, D-4630 Bochum-Querenburg, Federal Republic of Germany

Received 3 October 1984; accepted 15 October 1984

The CGTO integral evaluation, SCF, SCF-gradient, integral transformation, and MR-CI (SD) steps of the COLUMBUS system of programs have been adapted for the CYBER 205. A description is given of our efforts and the partly heavy modifications necessary to exploit the potential of this supercomputer and to avoid its shortcomings. Typical timings are reported, vector and scalar performance are compared.

I. INTRODUCTION

Supercomputers are becoming available to an increasing extent to quantum chemists as to other scientists. At present there are mainly the products of two companies, Cray Research and Control Data Corporation, which have been installed at a number of computation centers. The considerable interest they have found by (potential) users is documented by the fact that various conferences about supercomputers have been organized and that scientific journals devote an increasing amount of space to reviews and specialized reports on this topic. Detailed descriptions of the structure and the potential of supercomputers and comparisons of the respective performance have been published by Hockney and Jesshope.[1] These authors also discussed in detail the suitability of various algorithms, e.g., for performing matrix operations, for vectorization. Despite the potentially large speed of supercomputers for a variety of operations it is not yet clear what overall efficiency can be achieved for large, more complex and more strongly structured programs.

Quantum chemistry programs for *ab initio* electronic structure calculations may be considered as examples. A typical program package [for SCF, MC-SCF, and MR-CI(SD) calculations] involves about 100,000 lines of Fortran code. The bulk of computing is not concentrated in very few kernels, but is rather scattered over various program sections, and IO problems connected with the handling of large data sets, e.g., two-electron integrals, are of importance. Existing codes are, furthermore, in part, highly polished for (pipelined) scalar processers and take advantage of the fact that "near zeros" need not be computed. For these reasons it is a crucial question for the quantum chemistry community whether or not to invest a lot of man power and money for adapting and rewriting codes for the CRAY or CYBER 205, to give an example.

Several *ab initio* programs have been transferred to supercomputers. However, explicit reports are still scarce. The most detailed experience has been accumulated by the Daresbury group[2,3] for the CRAY 1. Vector processing capabilities are also offered by the FPS 164 of Floating Point Systems. Although the FPS 164 is about one order of magnitude slower than genuine supercomputers, it appears to pose quite similar problems to users. Various quantum chemistry groups now have access to the FPS 164. An extensive study including linear algebra benchmark tests and experience with

Journal of Computational Chemistry, Vol. 6, No. 3, 200-208 (1985)

CCC 0192-8651/85/030200-09$04.00

adapting the COLUMBUS program package and a GVB program has been published by Dunning and co-workers.[4]

In the present work we report our experience with the CYBER 205. The COLUMBUS program system[5-8] has been adapted and in part heavily modified to take advantage of the vector processing capabilities of this machine. Since we intended to perform mainly large scale CI calculations, we concentrated our efforts on the integral transformation and the direct CI module which are the most time consuming steps. We will not go into details too much since this could obscure rather than clarify the points we want to make. We rather follow the example of Dunning and co-workers[4] and present general considerations and conclusions. We will further report timings for test cases in comparison with those published by other groups.

II. GENERAL INFORMATION

A CDC CYBER 205—single pipe, 1 MW main storage, 2 GB disk storage, two channels—was installed at the University of Karlsruhe in summer 1983 (the machine was equipped with two pipes and the second pipe was recently switched on temporarily for test purposes). This report describes our experience with the implementation and vectorization of the COLUMBUS package, which includes the following programs: integrals over CGTOs,[5,6] SCF, integral transformation, and a GUGA multiple reference direct CI(SD).[7,8] Moreover, the following features were added to the original system: routines to compute first-order intermolecular SCF exchange energies,[9] an SCF gradient program was interfaced to the package, the CI(SD) was modified to allow for coupled pair type calculations within the recently developed coupled pair functional method (CPF),[10] a program to compute the first-order density matrix for MR-CI(SD) and CPF wave functions, and connected therewith the computation of expectation values (interfacing the POLYATOM programs extended for f AOs) such as electron density, multipole moments, static potentials, field gradients, and various population analyses.[11,12] Connected to the COLUMBUS system of programs is an MCSCF program.[13] This has so far not been transferred to the CYBER 205, mainly because we are awaiting a new release from R. Shepard. MCSCF computations can be done, of course, on the SIEMENS computer S 7880 (equivalent to the FUJITSU M 200), and vectors are then transferred to the 205 for MR-CI(SD) calculations.

In the following we discuss most of these programs under the aspects of vectorization in connection with their implementation on the 205. Several program steps (e.g., construction of the distinct row table and the formula tape, calculation of expectation values of one-electron properties) take only a negligible amount of the total time. No attempt was made to increase efficiency for these cases, which will not be discussed further here.

III. DETAILS

A. Simple Aspects of Vectorization

The CYBER 205 offers powerful vector processing capabilities for operations such as

$$\text{dyadic:} \quad C(I) = A(I) \ddagger B(I) \tag{1}$$

$$\text{triadic:} \quad C(I) = A(I) \ddagger B(I) \ddagger b \tag{2}$$

where the index I must be incremented in steps of one

$$I = 1, \ldots, N \tag{3}$$

and b denotes a scalar. These operations are performed at the vector processing speed R_v, which depends on the vector length N, the number of pipes n ($n = 1, 2$, or 4), and the number of start-up cycles s in the following way

$$R_v = nN/(ns + N) * \begin{Bmatrix} 50 \text{ dyadic} \\ 100 \text{ triadic} \end{Bmatrix} \text{ MFLOPS} \tag{4}$$

where 1 MFLOPS = one million floating point operations per second. For large N this means a sustainable speed of $50n$ and $100n$ MFLOPS for dyadics and triadics, respectively. This has to be compared with the scalar performance R_s which is

$$R_s \approx 5 \text{ MFLOPS} \tag{5}$$

for the 205 and rather well written programs. A straight Fortran code of eqs. (1) or (2) runs at $\approx$ 3 MFLOPS, which is improved by loop unrolling to 7 and 12 MFLOPS for dyadics and triadics, respectively.

The effective number of start up cycles s is not a well-defined quantity. It includes the bare start up of vector operations ($\approx$50 for dyadics and $\approx$80 for triadics), but other tasks have to be performed in addition such as loading of registers, loop organization (if vector instructions are in a loop), etc. For triadics we have measured effective start-ups

$$s \approx 130 \text{ (cycles)} \quad (6)$$

Now it will rarely be possible to achieve complete vectorization of a program. Let us introduce the degree of vectorization X, $0 \leq X \leq 1$, which is the fraction of total number of operations occurring in vector instructions. Assuming a rather uniform speed of the scalar and of the vector parts one gets the following (approximate) expression for the average speed $\overline{R}$

$$\overline{R}=R_v*R_s/[(1-X)R_v+XR_s] \quad (7)$$

One may now insert (4)–(6) into (7) to get a rough idea of the dependence of $\overline{R}$ on X, n, and the (average) vector length N. Since the best vector performance is obtained for triadics, one will clearly try to use mainly triadics, and we will employ the corresponding eq. (4) to obtain

$$X = 0.9,\ n = 1,\ N = 65 : \overline{R} \approx 20 \text{ MFLOPS} \quad (8)$$

$$X = 0.9,\ n = 1,\ N = 260 : \overline{R} \approx 30 \text{ MFLOPS} \quad (9)$$

$$X = 0.9,\ n = 2,\ N = 260 : \overline{R} \approx 35 \text{ MFLOPS} \quad (10)$$

Two conclusions are obvious: (i) In order to exceed the speed of a polished scalar code appreciably, say by a factor of four, one needs already a rather high degree of vectorization, $X \approx 0.9$, and an average vector length $N \approx 65$. (ii) To run the 205 at average speeds $\overline{R} > 50$ MFLOPS requires almost perfect vectorization, $X \gtrsim 0.95$, and long vector lengths, $N \gtrsim 260$. Only in this case is a two or four pipe installation significantly faster than a single pipe machine. Our experience with the supercomputer CYBER 205 is otherwise in many respects similar to that described by Dunning and co-workers[4] for the FPS 164:

(i) Avoid IF statements
(ii) Avoid GO TO statements
(iii) Avoid calls to subroutines in innermost loops by putting the corresponding code in-line.
(iv) If vectorization is not possible try to use "stacklib routines," a set of highly polished routines for various tasks provided by the system.

This helps not only to vectorize, but also speeds up scalar processing for the highly pipelined processors of supercomputers. A number of minor changes were made in various programs to obey these rules, which may easily double the scalar speed of the corresponding section.

It is further very important to improve IO in order to reduce residence time and IO charge. This implies the use of asynchronous IO—by means of the Q7BUFIN and Q7BUFOUT routines of CDC systems—and to make buffers as large as possible.

The compiler offers an option for automatic vectorization. However, we have decided to use explicit vectorization since not all vectorizable loops are vectorized, automatic vectorization may be counterproductive (short loops), and vector lengths have to be checked ($N \leq 2^{16} - 1$) anyway. Explicit vector statements further greatly increase the readability of programs.

B. The AO Integral Program

This is Pitzer's version[6] of Dupuis' HONDO[5] program which computes integrals over symmetry adapted CGTOs for D_{2h} and its subgroups. The program can handle up to g functions and always transforms to $5d$, $7f$, and $9g$ components in connection with the automatic input generator. An analysis—by means of the SPY system routine—revealed that 40–60% of the CPU time was required for the transformation step (symmetry and reduction of d-, f-, and g-AOs), the remainder was about equally distributed between overhead and various other routines, construction of the auxiliary I_X, I_Y, and I_Z integrals,[5] and their processing to compute the actual integrals. Attempts to vectorize the latter steps were counterproductive at first but we finally succeeded to vectorize the processing of I-integrals with the aid of gather operations.

The transformation of integrals over AOs to those over SAOs was rewritten for the most important case involving four different shells. This transformation may be written—as it was in principle in Pitzer's original version—as

$$(ab\,|\,cd) = \sum_{\mu\nu}\sum_{\kappa\lambda} CX(ab,\mu\nu)\,(\mu\nu|\,\kappa\lambda)\,CX(cd,\kappa\lambda) \qquad (11)$$

where μ, ν, κ, λ denote AOs and run over the orbitals which constitute a shell, and a, b, c, d denote the corresponding SAOs. The CX are obtained beforehand from coefficients defining SAOs in terms of AOs. Equation (11) has the form of a matrix transformation with $(\mu\nu)$, (ab), etc., as effective indices, and was coded in vectorized form as two successive matrix multiplications.

Efficiency of the integral program was further improved by moving tests on near zeros out of innermost loops (over contractions), i.e. by checking whether or not any of the integrals occurring in the contraction exceeds a threshold.

This results in altogether modest improvements in speed which was doubled as compared with the original scalar version. We note that in the present program version the transformation from AO to SAO integrals again requires ≈50% of CPU time.

C. SCF-Gradient Program

This program is based on Dupuis' HONDO routines as provided by Schaefer and Pitzer. It exploits symmetry in avoiding the computation of symmetry redundant integral derivatives. The check on near zeros for a whole batch of integrals was implemented as for the integral program. The effect was much larger in this case: CPU times went down to 37% in fortunate cases. In all probability this is due to the fact that logics is relatively slow on supercomputers and the gradient routines involve more logics than the integral programs.

Since integral derivatives are multiplied by corresponding SCF two-particle density matrix elements which are known beforehand, we further introduced a test

$$\text{DENMAX} * \text{INTTHR} > \text{TOL} \qquad (12)$$

where DENMAX is the maximum of two-particle density elements for the given batch (quadruple of shells) and INTTHR is the estimate for integral derivative contributions (INTTHR = e^{-G}, in the nomenclature of Rys, Dupuis, and King[5]). With the choice TOL = 1.E-9, the gradient is affected by 1.E-6. The check (12) further reduced CPU time by an additional 15 to 30%, and the total gain in performance is up to a factor of 4.

D. SCF Program

The COLUMBUS SCF routines have been heavily modified to improve convergence. Here we have relied mainly on our previous experience with level shifting,[14] damping, and the choice of open shell Fock operators, basically since we were accustomed to these features. More important, we have implemented Pulay's DIIS[15] convergence acceleration procedure. Matrix routines are vectorized but typically 80–90% of the CPU time is required by the construction of the Fock operator from two electron integrals. Attempts to vectorize the latter have been counterproductive so far (except for unpacking of labels of two-electron integrals which requires a considerable amount of time) as compared with scalar code with loop unrolling. However, an SCF calculation is IO bound anyway and the user is mainly charged for IO requests. To reduce IO charge we read a buffer of 24 small pages (1 small page = 512 words of 64 bits). To further reduce IO, small integrals are stored on disk in half precision.

E. Integral Transformation

The four-index transformation is done out of core or in-core if transformed integrals can be kept in main storage for the corresponding symmetry block. The in-core code uses Elbert's loop structure,[16] which was easily vectorized throughout using triadic operations. However, vector lengths are mainly N (≈50% of cases), in the remaining cases N^2 or N^3, where N stands for the number of functions of a given symmetry. No major effort was made to improve speed since this case requires little time anyway. The out of core transformation was heavily modified to achieve long vector lengths. The algorithm used now is based on the following kernel for the transformation of one index, $\lambda \rightarrow L$:

```
LOOP              L
ZERO              (μν|κL)   for all (μνκ)
LOOP              λ
VECTOR LOOP (μνκ): (μν|κL) =
                        (μν|κL) + (μν|κλ)
                        * U(λ,L)
END LOOP          λ,L
```

L denotes here an MO, and μ, ν, κ SAOs, and $(\mu\nu\kappa)$ the corresponding combined index. The crucial step in the above "one-index" transformation is a triadic operation — asymptotic speed $100n$ MFLOPS for n pipes — with vector length N^3, N being the number of SAOs for a given irreducible representation. Storage considerations limit the vector length — but without affecting the performance. The transformation of the next index ($\kappa \rightarrow K$) requires a reordering $(\mu\nu|\kappa L) \rightarrow (\mu\nu|L\kappa)$, which is done by means of periodic gather or scatter operations. This reordering would be unnecessary for a CRAY since this computer has a more general vector notation which allows constant index increments $\neq 1$, whereas consecutive storage is required for the 205. The just described algorithm is not used for the second transformation step ($\kappa \rightarrow K$) if κ and λ (and, hence, K and L) have the same symmetry. It was found easier to exploit $K \leq L$ and use a scalar product algorithm with vector length N. The scalar product (Q8SDOT) is relatively fast on the 205 (independent of the number of pipes): Asymptotic speed is 100 MFLOPS, a realistic start up time is $\approx$130 cycles. The transformation itself, especially the out of core part, appears to be efficient. However, the algorithm requires ordering of input integrals, and other tasks are performed within this step, such as the construction of the frozen core hamiltonian. Thus less than 50% of CPU time usually goes into the transformation itself, as compared with more than 80% for a scalar code.

The sorting steps are not vectorized, but their performance has been improved by a factor of 2, as compared with the original version, by unrolling of loops and coding short subroutines in line (a CALL requires about 300 cycles = 6 μs). The sorting algorithm is IO bound anyway, and IO charge cannot be reduced by simply increasing the buffer size, since the latter is determined by the available primary storage and the total number of integrals.

The situation found here probably occurs more often: On a scalar computer the N^5 step of transforming the integrals normally dominates N^4 steps like sorting. After efficient vectorization of the most important kernels the so far unimportant steps — or IO — become the new bottlenecks.

The problem in the present case is that the SAO integrals have to be sorted for an efficient transformation and transformed integrals are also reordered in order to prepare for the CI. For a given ordered quadruple of MO labels ($L \geq K \geq J \geq I$) the corresponding triple of integrals $(IJ|KL)$, $(IK|JL)$, and $(IL|JK)$ is collected. This facilitates the subsequent CI step, since the four-external integrals (i.e. all labels refer to external integrals) actually enter only in the following combination

$$K^{\pm}_{IJ,KL} = [(IK|JL) + (IL|JK)] * \{1/\sqrt{2} \quad \text{if } I = J \text{ or } K = L\} \tag{13}$$

(the case $I = J$ and $K = L$ enters diagonal CI matrix elements only and is treated separately), and from the three integrals one gets the corresponding three $K^{\pm}$ elements. These quantities are obtained directly in the final sort and stored as K^+ and K^- on disk, with minor modification described in the next section. This increases disk storage requirements (as compared to storing nonredundant triples), but these are still largest for the first sort step of SAO integrals.

This procedure has been chosen since it relieves the CI step from logics, which is now done only once instead in each CI iteration. It could increase the amount of data transfer from disk (as compared with the processing of nonredundant integrals), but for larger calculations (where the CI and the residual vector cannot be kept in main storage simultaneously) this is actually not the case since X and W paths, referring to K^- and K^+, have to be treated separately anyway by means of a "multiple pass" procedure. A similar technique is used for the treatment of three-external integrals.

F. Direct Singles and Doubles CI

The direct CI program consists basically of parts which process the diagonal elements of the CI matrix, the all-internal, and one-external to four-external two-electron integrals. Vectorization is straightforward as far as diagonal CI matrix elements, all-internal, and one-external integrals are concerned, but these steps require little effort anyway. Most time consuming is usually the processing of two-external and then that of four-external integrals, which will now be discussed in some more detail. The three-external integrals are treated in a similar way as the four-externals.

The processing of two-external integrals was completely redesigned to achieve a matrix formulation as suggested in detail by various authors.[3,17-20] This was possible without touching the basic structure of the program. These modifications made vectorization easy since one has to deal with matrix multiplications only and reduces the code (length of machine code of corresponding routines as produced by the Fortran compiler) to ≈30%.

It appears that the 205 is not well suited for this sort of problem, i.e. matrix multiplications. The latter is coded in outer product form[1] in using triadics. Since the vector length N is the number of external MOs per symmetry, typically between 10 and 70, we get from eq. (4) a performance between 7 and 30 MFLOPS.

The CRAY1 (one cycle = 12 ns) performs matrix multiplications at a maximum speed of ≈130 MFLOPS, and 65 MFLOPS are reached at $N \approx 7$.[1] This implies that the CRAY1 performs the matrix multiplications occuring in a CI(SD) at ≈100 MFLOPS, roughly a factor 5 faster than the 205.

The contributions of four-external integrals to the residual vector can be written in the following way

$$\sigma_{IJ} = \sigma_{IJ} + C_{KL} K_{IJ,KL}, \qquad (IJ) = 1, \ldots, (KL) \tag{14a}$$

$$\sigma_{KL} = \sigma_{KL} + C_{IJ} K_{IJ,KL}, \qquad (IJ) = 1, \ldots, (KL) - 1 \tag{14b}$$

where σ and C refer to the same internal part (internal walk) of the corresponding CSFs, and I, J, K, L run over the complete external space, with K and L as outer loops:

$$I \le J, \qquad K \le L, \qquad (IJ) \le (KL) \tag{15}$$

The matrices $C = (C_{IJ})$ are symmetric or antisymmetric if external orbitals are coupled to a singlet or triplet, respectively. $K^{\pm}$ is defined in eq. (11), and K^{+} is used for singlet, K^{-} for triplet couplings, i.e. W and X paths, respectively.

Formulas (14a, b) are well suited for the CYBER 205, provided the K-supermatrices are ordered canonically. In this case (14b) is a scalarproduct (asymptotic speed 100 MFLOPS), (14a) a triadic operation, and the vector length is in the order of N^2, N = dimension of external space. However, the construction of K is easiest if a triple of K's is obtained with indices running as $L \ge K \ge J \ge I$, as in the original COLUMBUS programs. Since the processing of four-externals is not the rate determining step and since we wanted to avoid another sort step, we decided to use the following procedure, which allows for some vectorization, at least.

The K supermatrix, dropping the $\pm$ label which is irrelevant at present, is stored as

$$A_{IJ}^{KL} = K_{IK,JL}, \qquad I \le J \tag{16}$$

$$A_{JI}^{KL} = K_{IL,JK}, \qquad I < J \tag{17}$$

$$B_{IJ}^{KL} = K_{IJ,KL} \tag{18}$$

where

$$I \le J \le K \le L \tag{19}$$

A and B are then matrices distinguished by the labels, K, L, and I, J as running indices. A is a full matrix whereas B is (anti) symmetric and stored in lower triangular form.

The summations of eqs. (14a) and (14b) are then basically—i.e. up to the special cases where some indices are identical—broken into the following pieces, with outer loops $L \ge K$.

$$\sigma_{KL} = \sigma_{KL} + \sum_{IJ=1}^{KL-1} B_{IJ}^{KL} C_{IJ} \tag{20a}$$

$$\sigma_{IJ} = \sigma_{IJ} + B_{IJ}^{KL} C_{KL}, \qquad IJ = 1, \ldots, (KL) \tag{20b}$$

$$\sigma_{JL} = \sigma_{JL} + \sum_{I=1}^{K} A_{IJ}^{KL} C_{IK}, \qquad J = 1, \ldots, K \tag{21a}$$

$$\sigma_{IK} = \sigma_{IK} + A_{IJ}^{KL} C_{JL}, \qquad I, J = 1, \ldots, K \tag{21b}$$

(20a) and (21a) are coded as scalar products with vector length (KL) and K, (20b) and (21b) as triadic operations with corresponding vector length. In the original definitions of Shavitt[8] the contributions (20a, b) correspond to the loop cases (3a, b) and (21a, b) to a combination of the loop cases (1a, b) and (2a, b).

The programs exploit symmetry within D_{2h} and its subgroups. This clearly results in additional overhead. The vector length is then typically N^2 for (20a, b) and N for (21a, b), where N denotes the number of external MOs per symmetry.

We have further rewritten the Davidson diagonalization part (to reduce IO), have implemented the new CPF method[10] to include

cluster corrections and to achieve size consistency, and added routines to compute the first order density matrix.

IV. TYPICAL TIMINGS

In Table I we report timings for the integral, SCF, and gradient calculations, in Table II for the transformation and CI steps. Six large pages (64 K words each) were required in most cases. The programs should not be run with less than 4 pages, but 4 or 5 are possible although this may result in an increase in page faults (depending on the actual case, e.g. basis set size).

The SCF calculations were partly done without providing starting vectors. The iteration cycle was usually terminated if $\Delta E < 10^{-7}$ and $\|\Delta F\| < 10^{-4}$ ($\|\Delta F\|^2 = \Sigma_{\nu<\mu} |\Delta F_{\nu\mu}|^2$, in an orthonormal basis). The latter requirement is

Table I. Timings for integral and SCF calculations (in seconds)

Molecule	O_3	$(H_2O)_2$	Cu_2	Cu_2	$(N_2)_2$	$ClCH_2SiCl_3$
Symmetry	C_{2v}	C_s	D_{2h}	D_{2h}	C_{2v}	C_s
Basis/size[a]	*A*/69	*B*/68	*C*/144	*D*/138	*E*/116	*F*/146
INT CPU	57	98	558	652	320	930
INT total[b]	77	123	767	867	415	1120
SCF CPU	16	27	82	64	91	168
SCF total[b]	45	65	259	200	250	378
SCF iterations[c]	14	12	23	21	14	9
Total INT + SCF[b]	122	188	1026	1067	665	1498

[a] Basis sets: *A*: (10, 6, 1)/[6, 4, 1],
B: (11, 7, 1; 6, 1)/[5, 4, 1; 3, 1],
C: (16, 11, 6, 3)/[10, 7, 4, 3],
D: (14, 11, 6, 3, 1)/[8, 6, 4, 2, 1],
E: (12, 6, 2)/[7, 4, 2],
F: Si, Cl: (11, 7, 1)/[6, 4, 1], *C*: (9, 5, 1)/[5, 3, 1]
H: (5, 1)/[3, 1].

[b] Total charge, normalized such that 1 CPU second contributes 1 unit. The remaining charge accounts for IO, main storage requirements, page faults, etc.

[c] All calculations without starting vectors, except for the $ClCH_2SiCl_3$ molecule.

Table II. Timings, in seconds, for transformation and CI(SD) or CPF calculations.[a]

Molecule	O_3	O_3	Cu_2[b]	Cu_2[c]	$(H_2O)_2$	$(N_2)_2$	$(N_2)_2$
Symmetry	C_{2v}	C_{2v}	D_{2h}	D_{2h}	C_s	C_{2v}	D_{2h}
State	$^1A_1$	$^3B_2$	^1A_g	^1A_g	$^1A'$	$^1A_1$	^1A_g
N_i[d]	9	10	11	11	8	10	10
N_e[e]	54	53	113	109	56	98	98
N(SD)[f]	31,440	37,166	98,594	91,584	52,265	128,527	63,993
Trafo CPU	28	28	242	202	46	195	135
Total charge[g]	62	62	539	449	112	766	296
CI iteration	7	10	31	29	13	41	19
Number of it	7	6	15	15	6	8	9
Type of calc.	CI	CI	CPF	CPF	CI	CPF	CPF
Total CPU	50	60	470	437	78	329	173
Total charge[g]	136	155	1091	1000	194	884	397

[a] Basis sets as in Table I (see also explanations in text).
[b] Basis C of Table I.
[c] Basis D of Table I.
[d] Number of internal MOs.
[e] Number of external MOs.
[f] Number of SD configuration state functions.
[g] See footnote b of Table I.

more stringent, ΔE is often 10^{-10} or less, which is in the order of rounding errors.

The CI(SD) typically gains one decimal in the energy per iteration. For ozone and the water dimer E is in error by $\approx 10^{-6}$ after 5 to 6 iterations. Convergence is poorer for CPF calculations where 8 [$(N_2)_2$] to 15 (Cu_2) iterations are required to push the error below 10^{-6} in the energy.

Inspection of the tables reveals that the integral calculation is CPU bound ($\approx$80% of charge is for CPU) the SCF step is clearly IO bound (30% CPU), whereas the transformation and CI are roughly balanced (40–50% CPU). For all cases listed, the integral plus SCF part requires about the same effort as the transformation plus CI(SD) or CPF calculation. However, the CI will dominate in multiple reference calculations, since the CI time increases roughly linearly with the number of CSFs included: One iteration requires $\approx$30 seconds per 100,000 CSFs. This is about 3 times more than the CRAY 1 programs of Saunders and van Lenthe[3] or Werner,[20] which is expected by means of the considerations given in section IIIF. The present timings for the transformation step appear to be shorter than those published by Saunders and van Lenthe,[3] e.g., 28 s for O_3 (Table I) vs. 49 s, or 46 s vs. 58 s for $(H_2O)_2$.

SCF gradient calculations require only slightly more effort than the preceeding integral plus SCF steps. For $ClCH_2SiCl_3$, as a typical representative of a series of calculations on similar molecules, the gradient step required 1540 CPU seconds and a total charge of 1650 units, as compared to 1498 units for integrals plus SCF (Table I).

V. CONCLUSIONS

After we knew we would get access to a CYBER 205 we decided firstly to get the programs running at all — which was not so easy — and then to vectorize the modules which are most expensive: the transformation and the direct CI. Although this required partly heavy modifications of the code, it was not too difficult. The integral transformation algorithm is basically simple, and it was relatively easy to achieve long vector lengths for the most important cases. For the CI steps we had experience with a matrix formulated procedure,[18] which should be ideal for vector processors. However, the 205 is relatively slow for most matrix operations where the vector length is only N, the number of rows or columns of the corresponding matrix. Since N is typically between 10 and 70 in the matrix formulated CI, the corresponding steps are dominated by start-up times, and we have been basically fighting start-ups.

This exemplifies one of our experiences with the 205: This is a very fast supercomputer if long vector lengths can be achieved and $50n$ MFLOPS for dyadic and $100n$ MFLOPS (n = number of the pipes) for triadic operations are then easily achieved in Fortran codes. Owing to the impressive transfer rate between main storage and CPU (up to 800 million words/s) one is not bound by data transfer as long as one works in core only. Besides our attempts to cut down start-ups we mainly had to reduce IO charges and residence times by means of concurrent IO and using large buffers. In our opinion this problem should be taken care of by the system which fails in this respect (like for most computers): Fortran IO is very expensive both in CPU and IO charge.

For the integral, SCF, and gradient programs we made an attempt for vectorization — but could not afford to rewrite these parts completely — which was only partly successful. Although parts of the present code can certainly be improved, e.g., the sorting of integrals, we do not see at present how the efficiency of any of the programs discussed in this article could be improved by a factor of two — both in CPU or total charge — without a complete restructuring, for which we do not have solid ideas at the moment.

The decisive point in a comparison between supercomputers and "scalar" computers is the price/performance ratio. It is very difficult to talk about prices but when the SIEMENS S 7880 and the CYBER 205 were installed in 1983, the 205 was about twice as expensive as the S 7880. We can compare directly the CPU times of the original COLUMBUS version and the present 205 version of programs on both machines. CPU times for the S 7880 are 0–50% larger than for the 205 in scalar mode. The ratio between CPU and total charge is roughly comparable on both computers.

CPU (S 7880)/CPU (CYBER 205)	
Integrals	2–4
SCF	2
Gradient	3–5
Transformation	≥4.5
CI	≥5
typical total CI	≥4

The "≥" signs are used since we could only compare relatively small computations and the 205 usually "wins" for larger basis sets. These gains are mainly due to vectorization except for the gradient step. The numbers clearly indicate the power of the 205 although it would be even better if start-ups were shorter. The 205 is a potentially more powerful supercomputer than we have been able to exploit. However, about 20 years of experience have gone into developing efficient codes for pipelined scalar computers and the virtually complete restructuring of algorithms — and thinking of scientists — required in order to take advantage of the features of supercomputers will probably take some time. The main problem — at least in quantum chemistry — appears at present to avoid or at least to reduce or optimize IO, which may be achieved by solid state disk technology, to give an example.

One of us (H. L.) thanks the Humboldt Foundation for a travel grant. This work was supported by the Österreichischer Fonds zur Förderung der wissenschaftlichen Forschung (P4965), the DFG (Ah 9/13), and in part by the Fonds der Chemischen Industrie. We thank the Rechenzentrum der Universität Karlsruhe for its excellent service.

References

1. R. W. Hockney and C. R. Jesshope, *Parallel Computers,* Hilger, Bristol, 1981.
2. V. R. Saunders and M. F. Guest, *Comput. Phys. Commun.*, **26,** 389 (1982).
3. V. R. Saunders and J. H. van Lenthe, *Mol. Phys.*, **48,** 923 (1983).
4. (a) R. Shepard, R. A. Baird, R. A. Eades, A. F. Wagner, M. J. Davis, L. B. Harding, and T. H. Dunning, *Int. J. Quantum Chem. Quantum Chem. Symp.*, **17,** 613 (1983); (b) R. A. Baird and T. H. Dunning, *J. Comput. Chem.*, **5,** 44 (1984).
5. M. Dupuis, J. Rys, and H. F. King, *J. Chem. Phys.*, **65,** 111 (1976); J. Rys, M. Dupuis, and H. F. King, *J. Comput. Chem.*, **4,** 154 (1983).
6. R. M. Pitzer, *J. Chem. Phys.*, **58,** 3111 (1973).
7. H. Lischka, R. Shepard, F. B. Brown, and I. Shavitt, *Int. J. Quantum Chem. Quantum Chem. Symp.*, **15,** 91 (1981).
8. I. Shavitt, Annual Report to NASA Ames Research Center, June 1979.
9. H.-J. Böhm, R. Ahlrichs, P. Scharf, and H. Schiffer, *J. Chem. Phys.*, **81,** 1389 (1984), and references therein.
10. R. Ahlrichs, P. Scharf, and C. Ehrhardt, *J. Chem. Phys.*, in press.
11. R. S. Mulliken, *J. Chem. Phys.*, **23,** 1833 (1955).
12. R. Heinzmann and R. Ahlrichs, *Theor. Chim. Acta,* **42,** 33 (1976).
13. R. Shepard, I. Shavitt, and J. Simons, *J. Chem. Phys.*, **76,** 543 (1982).
14. M. F. Guest and V. R. Saunders, *Mol. Phys.*, **28,** 819 (1974).
15. P. Pulay, *Chem. Phys. Lett.*, **73,** 393 (1980).
16. S. Elbert, Report on the NRCC workshop on Numerical algorithms in chemistry: Algebraic methods, 1978, p. 129.
17. H.-J. Werner and E.-A. Reinsch, *J. Chem. Phys.*, **76,** 3144 (1982), and references therein.
18. R. Ahlrichs, in *Methods in Computational Molecular Physics,* G. H. F. Diercksen and S. Wilson Eds., Reidel, Dordrecht, 1983.
19. W. Meyer, R. Ahlrichs, and C. E. Dykstra, in *Vectorization of Advanced Methods for Molecular Electronic Structure,* C. E. Dykstra, Ed., NATO ASI, Reidel, Dordrecht, 1984.
20. H.-J. Werner and E.-A. Reinsch, see ref. 19.