Theor Chim Acta (1987) 72: 149-173

Theoretica Chimica Acta

Strategies to vectorize conventional SCF-CI algorithms

Nobuhiro Kosugi

Department of Chemistry, Faculty of Science, The University of Tokyo, Hongo, Tokyo 113, Japan

(Received September 3, 1986, revised June 26/Accepted July 3, 1987)

Recent high-performance computers, especially supercomputers, achieve very high-speed operations but bring about serious I/O problems in quantum chemical computations. Strategies to vectorize conventional SCF-CI algorithms are discussed relating to the I/O problems. The conventional SCF-CI algorithm which is proposed here reduces I/O processing by eliminating all sorting routines and redundant integral files and generates directly nonzero and nonredundant *PK* integrals with a vectorizable canonically-ordered list. The new implementation has been undertaken and successfully realized as a program system named GSCF3. The vector to scalar acceleration rate of GSCF3 on the HITAC S-810 are as follows: 2.5 ~ 5 in the AO integral evaluation, 5 ~ 12 in the SCF calculation, 15 ~ 30 in the four-index integral transformation, 10 ~ 20 in the CI matrix diagonalization, and overall 5 ~ 10 through SCF-CI.

Key words: Conventional SCF-CI — I/O bottleneck — Vectorization of indirect addressing — Vectorizable canonically-ordered *PK* integrals — Four-index transformation

1. Introduction

The appearance of supercomputers (vector processors) has brought rapid growth of computational capabilities. Experience in adaptation of quantum chemical computations to the supercomputers is being accumulated in the quantum chemistry community [1-5]. The characteristics of these machines differ so much from those of scalar machines that a thorough rethinking of the computational strategies is necessary. At this time the capabilities of compilers converting effective scalar codes to effective vector codes are still severely limited. It is still

a crucial decision for quantum chemists developing highly-polished scalar codes whether or not to completely rewrite computer codes for supercomputers.

Although recent computers, vector or scalar, attain *very* high-speed operations and enable *very* large-scale computations, handling of large data files is more cumbersome. It is because the word "speed" means CPU speed of arithmetic and logical operations and memory speed of cache memory (register) and main storage (MS, central or "core" memory), but does not mean channel speed in data transfer between MS and auxiliary storage (external disk systems) [1-3]. That is, algorithms which are designed with a great effort to well balance CPU and I/O times on today's computers will always be I/O bound on tomorrow's computers.

There are some algorithms proposed to avoid the serious I/O problems in quantum chemical computations: direct SCF [6] and direct CI [7]. In the direct SCF approach [6], only the non-negligible two-electron integrals to the Fock matrix (or to changes in the Fock matrix) are re-evaluated in every SCF iteration. This approach has been primarily conceived for minicomputers with highly sophisticated CPUs but with much less developed I/O systems. It also enables very large-scale calculations using very extensive basis sets in which the storage of integrals in the conventional two-step (integral and SCF) procedure is prohibitive at any existing computer system. The same situation occurs in conventional CI calculations [7] as in conventional SCF calculations. The corresponding CPU bound and I/O bound steps are Hamiltonian matrix generation and iteratively solved matrix diagonalization, respectively. In the direct CI method, wavefunctions are constructed directly from molecular two-electron integrals without explicitly generating an intermediate Hamiltonian matrix. The above "direct" approaches have, however, several weak aspects. The direct SCF approach [6] is completely CPU-bound; therefore, the CPU requirement for the integral step is usually much larger than that for the SCF step (per iteration) and its effectiveness is heavily dependent on CPU and I/O performances inherent in available computer systems. In contrast, whatever type of computer we are using, the direct CI algorithm [7] is most promising. However, reduction of the configuration list based on individual configuration selection by perturbation theories on energy contributions and by restriction schemes on electron configuration types is not easily attained in the direct CI method without destroying the simple and inherently efficient matrix-multiplication structure.

There are some bright outlooks for hardware and software technology to reduce or eliminate disk I/O [5]. The latest semiconductor technology has resulted in high-performance computer systems, vector or scalar, with huge main storage (MS, e.g. 2 GB (giga byte) in the CRAY 2) and huge storage (e.g. 3 GB in the HITAC S-810/model 20) connected with MS through high-speed channels (e.g. 1.3 GB/s in the NEC SX-2). The latter is called Solid State Device or "Disk" (SSD) or Extended Storage (ES). Then, I/O times can be easily reduced or eliminated through use of greater amounts of MS and/or ES instead of magnetic disk systems, though the savings depend on how critical the memory resources

are. Another approach to reduce the amount of I/O time is to perform the disk I/O concurrently with either computations or other I/Os. The former is asynchronous disk I/O attained by rewriting so as to use double buffering, but is not effective to heavily I/O-bound jobs. The latter is parallel disk I/O (or disk stripe) which is attained by splitting a data file into several pieces on separate I/O channels and by processing the I/O channels concurrently. The parallel I/O technique can almost cope with heavily I/O-bound jobs, though total speed of the I/O channels is at most 50 MB/s (~3 MB/s × 16 parallel in the HITAC S-810 and M-680H) at the present stage of technology.

In the present work, a new implementation of "conventional" SCF-CI algorithms is proposed for adaptation to recent high-performance computer systems, especially to supercomputers. In Sect. 2, a basic design of the "new" conventional SCF-CI approach proposed is discussed relating to I/O problems brought about by the recent computers. In Sects. 3 and 5, vectorization of AO integral evaluation, Fock matrix generation and Hamiltonian matrix diagonalization are considered. In Sects. 4 and 6, how to arrange two-electron AO integrals into a canonically-ordered *PK* supermatrix and how to transform the four-index AO integrals to the MO integrals are proposed so as to satisfy the 'new' conventional SCF-CI formalism. In the last section, typical timings of the present algorithms on HITAC S-810 are shown and discussed. The computer program used has been under development for about a decade [8] and the current implementation is named GSCF3 [9].

2. A "new" conventional SCF-CI algorithm

A typical example of flow charts for conventional SCF-CI programs is shown in Fig. 1. In the integral program INT, two-electron integral g_{pqrs}'s are evaluated over contracted Gaussian-type basis functions (atomic orbitals, AOs) and are stored as the *g* file in external storage. To efficiently perform SCF calculations, the *PK* integral file is obtained by sorting and preprocessing an unordered AO integral list (the *g* file) [10]. Then, the *PK* file is read one or more times at every SCF iteration.

To proceed beyond Hartree-Fock calculations, we have to obtain two-electron

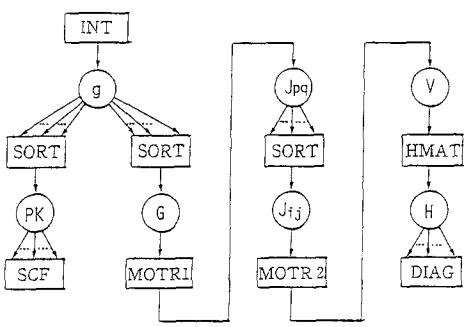

Fig. 1. A typical example of flow charts for conventional SCF-CI programs, where *circles* and *rectangles* denote data files and program parts, respectively

integrals V_{ijkl}'s over molecular orbitals X_i's (MOs) from the AO integrals G_{pqrs}'s through the following four partial summations:

(I) $T_{pqri} = \Sigma_s G_{pqrs} X_{is}$
(II) $J_{pqji} = \Sigma_r T_{pqri} X_{jr}$
(III) $U_{pkji} = \Sigma_q J_{pqji} X_{kq}$
(IV) $V_{lkji} = \Sigma_p U_{pkji} X_{lp}$.

In the current approaches [11], the four-index integral transformation consists of two steps. The first two-index transformation (MOTR1) $G_{pqrs} \rightarrow T_{pqri} \rightarrow J_{pqji}$ $(rs \rightarrow ji)$ for all p, q, i and j is followed by the second two-index transformation (MOTR2) $J_{pqji} \rightarrow U_{pkji} \rightarrow V_{lkji}$ $(pq \rightarrow lk)$ [11]. In advance of execution of MOTR1, we have to sort the unordered AO integral list of the g file into an ordered list suitable for MOTR1. In the resultant integrals $G_{pq}(rs)$ stored in the G integral file, pq is fixed and rs runs over AOs. Between MOTR1 and MOTR2, all the half-transformed integrals J_{pqji}'s are stored in external storage and transposed by sorting of $J_{pq}(ij)$ for all ij with fixed pq to $J_{ij}(pq)$ for all pq with fixed ij. Using the MO integrals V_{ijkl}'s, Hamiltonian matrix elements are constructed in the H file by the HMAT program. Finally, the Hamiltonian matrix is diagonalized to obtain eigensolutions. Matrix diagonalization problems are iteratively solved and the H file is read once at every iteration [12].

Through the conventional SCF-CI algorithms, we have to handle as many as seven huge integral files, though no more than two such files ever need to exist at the same time. The number of elements in the AO integral files, g, PK and G, is $Nao^4/8$ where Nao denotes the number of basis functions in SCF (AOs). The number of elements in the intermediate files, J_{pq} and J_{ij}, and in the MO integral file, V, are $Nao^2Nmo^2/4$ and $Nmo^4/8$, respectively, where Nmo denotes the number of basis functions in CI (MOs). The number of elements in the H file is $Nci^2/2$ where Nci denotes the dimension of the Hamiltonian matrix. Ideally the J_{pq} and J_{ij} files and two of the three AO integral files are unnecessary and should be eliminated. Cumbersome file handlings and huge storage requirements by them come from the three sorting routines, SORT. In order to avoid I/O bottlenecks which would be encountered in recent high-speed computers, we have to re-design a new "conventional" SCF-CI algorithm in which all the sorting routines are eliminated.

An ideal "conventional" SCF-CI algorithm is shown in Fig. 2, where we have only to handle three disk files of AO integrals (the PK file), MO integrals (the V file) and Hamiltonian matrix elements (the H file). In the following sections, how to realize this "new" conventional SCF-CI algorithm is described and discussed from the viewpoint of vectorization.

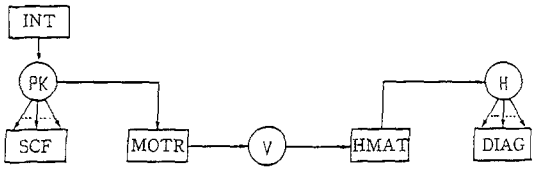

Fig. 2. A new algorithm for the conventional SCF-CI calculation proposed to overcome I/O bottlenecks brought about by recent "high-speed" computers

3. Vectorization of AO integral evaluation

In this section, some considerations on the "extrinsic" vectorization algorithm for the AO integral evaluation by the Daresbury group (Saunders et al.) [15, 16] are described.

The Pople-Hehre method [13] and the Gauss-Rys method [14] are widely employed for integration of *s* and *p* functions and of *s*, *p* and Cartesian *d* functions, respectively, using the batch (or shell) processing algorithm [13]. Saunders et al. have proposed the "extrinsic" vectorization algorithm [15, 16], in which the vector length of innermost loops is *MUMAX*, the number of nonzero integrals with different primitive combinations in a given integral batch. The "extrinsic" algorithm does not work so efficiently in vectorization of the Pople-Hehre method because of laborious preprocessing and FORTRAN overhead for vectorization (initializing DO loops).

On the other hand, in the Gauss-Rys method, further refinements for the "extrinsic" vectorization can be achieved over the original algorithm [15, 16]. In Figs. 3 and 4, the "orthodox" and the vector-adapted kernels of the Gauss-Rys code are shown, respectively, where N is the number of the Gauss-Rys quadrature points, and *NINT* the dimension of one integral batch (the number of AO integrals to be evaluated simultaneously in one integral batch). In the orthodox code (3a), there are two algorithms shown; one is a simultaneous multiplication among the auxiliary integrals *Xi*, *Yi* and *Zi*, the other is a two-step multiplication. The latter has an advantage when $MINT < NINT$. For example, when $MINT = 1296$, $NINT = 10000$, and all the basis functions belong to *spd* shells, the number of multiplications in the two-step algorithm (11 296) is about half of that in the simultaneous algorithm (20 000). Such a saving is also possible in the other orthodox code (3b) and the vector codes (4a) and (4b). It should be noted that main storage requirements can be reduced by replacing all the vectors of dimension 10 000 with those of dimension 1296 if neither *pd* nor *spd* shells are accepted (if the restriction of equal *s*, *p* and *d* exponents is removed).

There is another orthodox code shown in Fig. 3b, which can reduce main storage requirement of *TEMP* drastically from 10 000 to only 5 ($N = 5$ when all the functions are of *d*, *pd* or *spd* type). In Figs. 4a, b, the orthodox codes (3a) and (3b) are modified for vectorization, respectively, where # denotes the number of auxiliary integrals with different primitive combinations to be evaluated concurrently; in the code (4b), auxiliary integrals for different Gauss-Rys quadrature points as well as different primitive combinations are evaluated concurrently, though some additional preprocessings are needed for achieving this concurrency. The description of the codes in Fig. 4 is simplified for presentation; it is of course that DO 300 must account for the remainder using vector length less than # in the last loop. The case that # equals *MUMAX* (or, sometimes, the optimal vector length depending on computer systems) is ideal but the maximum limit of # is dependent on available main storage. In the code (4b), # can be set larger than in the code (4a) because of less main storage requirements, and what is still better, the vector length of innermost loops can be set relatively long even

a Orthodox code

```
dimension  :  TEMP(10000), X(81), Y(81), Z(81), COEF(256)
                          (  XY(1296)  )
           :  NZ(10000), NC(10000), G(10000)
           :  NX(10000), NY(10000)
           or MX(1296), MY(1296), MXY(1296), NXY(10000)

      DO  300  mu=1, MUMAX

      DO  100  i=1, N

    (obtain Xi, Yi, Zi)

      DO  110  ng=1, NINT
  110 TEMP(ng)=TEMP(ng)+X(NX(ng))*Y(NY(ng))*Z(NZ(ng))
or
      DO  111  mg=1, MINT
  111 XY(MXY(mg))=X(MX(mg))*Y(MY(mg))
      DO  112  ng=1, NINT
  112 TEMP(ng)=TEMP(ng)+XY(NXY(ng))*Z(NZ(ng))

  100 CONTINUE

      DO  200  ng=1, NINT
  200 G(ng)=G(ng)+COEF(NC(ng))*TEMP(ng)

  300 CONTINUE
```

b Another orthodox code (an example when $N=5$)

```
dimension  :  TEMPi, Xi(81), Yi(81), Zi(81)  ( XYi(1296) )  for all i

      DO  300  mu=1, MUMAX

    ( obtain  Xi, Yi, Zi  sequentially      for all i)

      DO  200  ng=1, NINT

    ( obtain  TEMPi=Xi(NX(ng))*Yi(NY(ng))*Zi(NZ(ng))      for all i)
                        or  XYi(NXY(ng))*Zi(NZ(ng))

  200 G(ng)=G(ng)+COEF(NC(ng))*( TEMP1+TEMP2+TEMP3+TEMP4+TEMP5 )

  300 CONTINUE
```

Fig. 3. Some modifications of the "orthodox" Gauss-Rys codes

when *MUMAX* is rather small, that is, basis functions are low contracted or partly uncontracted. When all the basis functions in one integral batch are uncontracted, we should employ the code (3a) or (3b). Both the codes (3a) and (3b) in which indirect addressing with the index vectors is used in the innermost loops are vectorized; however, their vectorization is not so efficient or, sometimes, even counter-productive because of frequent access to the same address (memory bank conflicts).

4. Direct generation of a canonically-ordered *PK* integral file

To satisfy the "new" conventional SCF-CI formalism, a canonically-ordered *PK* integral (supermatrix) file [17] is generated directly by the AO integral evaluation

a Vector-adapted code for (3a)

```
dimension  :  TEMP(#, 10000), X(#, 81), Y(#, 81), Z(#, 81), COEF(#, 256)
                                ( XY(#, 1296)    )

      DO  300  mu=1, MUMAX, #

      DO  100  i=1, N

    ( obtain  Xi, Yi, Zi  concurrently     for  # primitives)

      DO  110  ng=1, NINT
      DO  110  mt=1, #
  110 TEMP(mt, ng)=TEMP(mt, ng)+X(mt, NX(ng))*Y(mt, NY(ng))*Z(mt, NZ(ng))
                                          or  XY(mt, NXY(ng))*Z(mt, NZ(ng))
  100 CONTINUE

      DO  200  ng=1, NINT
      DO  200  mt=1, #
  200 G(ng)=G(ng)+COEF(mt, NC(ng))*TEMP(mt, ng)

  300 CONTINUE
```

b Vector-adapted code for (3b)

```
dimension  :  TEMP(5*#), X(5*#, 81), Y(5*#, 81), Z(5*#, 81), COEF(#, 256)
                          ( XY(5*#, 1296)  )

      DO  300  mu=1, MUMAX, #

    ( obtain  Xi, Yi, Zi  concurrently     for  # primitives & all i)

      DO  200  ng=1, NINT
      DO  110  nt=1, N*#

  110 TEMP(nt)=X(nt, NX(ng))*Y(nt, NY(ng))*Z(nt, NZ(ng))
                    or  XY(nt, NXY(ng))*Z(nt, NZ(ng))

      DO  200  mt=1, #
  200 G(ng)=G(ng)+COEF(mt, NC(ng))*(                    TEMP(5*mt-4)
     *                             +TEMP(5*mt-3)+TEMP(5*mt-2 )
     *                             +TEMP(5*mt-1)+TEMP(5*mt    ))

  300 CONTINUE
```

Fig. 4. Vector-adapted Gauss-Rys codes

program. First of all, to simplify its algorithm for presentation, an algorithm to evaluate AO integrals not batch-wise but individually is shown in Fig. 5, where indices p, q, r and s denote AOs, *Nao* the number of AOs, and *rs* a canonically-ordered index $[r, s]$ for packed symmetric matrices ($[a, b]=[b, a]=a(a-1)/2+b$, $a \geq b$). The quadruple AO loop structure, what is called Meyer's loop structure (as quoted in ref. [18]), generates nonredundant AO integrals G_{pqrs}, G_{psqr} and G_{prqs} sequentially, where their trivial and non-trivial redundancies are checked by the nature of identities between the four labels p, q, r and s and by whether or not a symmetry operation maps the four-AO label $\{pqrs\}$ ($p \geq q \geq r \geq s$) into a larger/smaller label $\{p'q'r's'\}$ (rearranged in descending order), respectively. The check whether or not $\{pqrs\}<\{p'q'r's'\}$ can already be performed partly in the outer loops ($p<p'$, $\{pq\}<\{p'q'\}$, $\{pqr\}<\{p'q'r'\}$) before all the indices p, q, r and

```
      DO   100   p=1, Nao

      DO   110   q=1, p
      DO   110   r=1, q
      DO   110   s=1, r

    ( symmetry check for the AO quadruplet {pqrs})

      evaluate Gpqrs
      evaluate Gpsqr if q.NE.r or  (p.NE.q and r.NE.s)      [case 1]
      evaluate Gprqs if q.NE.r and p.NE.q and r.NE.s        [case 2]
      Kp(q, rs)=Gpsqr+Gprqs
      Pp (q, rs)=Gpqrs-Kp(q, rs)/4
      Kp(s, qr)=Gprqs+Gpqrs                                 [case 1]
      Pp (s, qr)=Gpsqr-Kp(s, qr)/4                          [case 1]
      Kp(r, qs)=Gpsqr+Gpqrs                                 [case 2]
      Pp (r, qs)=Gprqs-Kp(r, qs)/4                          [case 2]

110 CONTINUE

    ( Here, we have PK integrals Pp(q, rs) & Kp(q, rs)
            in which   q=1 to p   &   rs.LE.pq)

      DO   120   q=1, p

      extract   Ppq(rs) & Kpq(rs)   from   Pp(q, rs) & Kp(q, rs)

              Npq=0
              DO   121   rs=1, pq
              IF(.NOT.SYM(rs).or.ABS(Ppq(rs))+ABS(Kpq(rs)).LT.eps)   GO   TO   121
              Npq=Npq+1
              RS(Npq)=rs
121           CONTINUE

              Mpq=Npq
              DO   122   rs=1, pq
              F(SYM(rs).or.ABS(Ppq(rs))+ABS(Kpq(rs)).LT.eps)   GO   TO   122
              Mpq=Mpq+1
              RS(Mpq)=rs
122           CONTINUE

      IF(.NOT.SYM(pq))   Npq=0

      write   Npq, Mpq
      write   Ppq(RS(n)), RS(n), n=1, Mpq
      write   Kpq(RS(n))        , n=1, Mpq

120 CONTINUE

100 CONTINUE

      SYM(rs)=.TRUE.      if the r'th and s'th AOs belong to the same IR
                          (except that the same AO contributes to different IRs)
      SYM(rs)=.FALSE.     if not
```

Fig. 5. Direct generation of a canonically-ordered PK integral file

s are specified [18, 19]. Then, we can easily obtain PK integrals with non-redundant AO combinations among p, q, r and s as shown in Fig. 5, where the PK integrals must be divided by the number of symmetry operations which map $pqrs$ into itself according to the Dacre scheme [19, 20]. When the inner three loops (DO 110) are completed for a specified p (DO 100), PK integrals Pp_{qrs} and Kp_{qrs} are all obtained under the conditions that $p \geq q$, r, s and the labels p, q, r and s are in canonical order ($p \geq q$, $r \geq s$, $[p, q] \geq [r, s]$). We can pick up PK integrals Ppq_{rs} and Kpq_{rs} in which pq is fixed and rs runs from 1 to pq (if there is no symmetry-redundancy). After checking whether or not any of the P and K integral sums exceeds a given threshold (eps), we can store in external storage only "nonzero" and nonredundant integrals with canonical indices RS at each pq combination of AO indices. Mpq denotes the number of the "full" P and K integrals to be stored for a specified pq combination, and Npq the number of the "purged" PK integrals where both the p'th and q'th AOs and both the r'th and s'th AOs are components of symmetry-adapted orbitals belonging to the same symmetry species (irreducible representation, IR), respectively, exclusive of the case that the same AO contributes to different IRs. The purged PK integral list is used in one-electron calculations (SCF) where one-electron operators (Fock and Fock-like operators, one-electron density matrices) should be totally symmetric with respect to molecular symmetry. The full PK integral list must be used in beyond-HF calculations where two-electron operators play an important role. This purging is originally designed and very effective for the symmetry-adapted PK integrals [15] and is also effective to the AO-based PK integrals as has recently been demonstrated by Bair [21]. Only RS, Mpq and Npq are needed as indexing information for the integral labels. In the algorithm as is shown in Fig. 5, MS (main storage) of Nao^3 words is required for Pp and Kp when $p = Nao$; in practice, we have only to have MS of Nao^2 words for Ppq and Kpq for a specified q by storing Pp and Kp in a work file of Nao^3 words (at maximum).

An algorithm to evaluate AO integrals batch-wise is shown in Fig. 6, where $Nshell$ denotes the number of shells and indices P, Q, R and S denote labels of shells [13]. Meyer's loop structure [18] is applied to the loop over shells. A shell with a label P consists of AOs with AO labels p from $pmin$ to $pmax$. Non-trivial redundancy in the integral evaluation due to molecular symmetry can be checked by the inequality relations between the original four-shell label $\{PQRS\}$ and the labels $\{P'Q'R'S'\}$ (rearranged in descending order) subject to symmetry operations within the framework of shell labels instead of AO labels. The check whether or not $\{PQRS\} < \{P'Q'R'S'\}$ can beforehand be performed partly in the outer loops more cheaply than the check whether or not $\{pqrs\} < \{p'q'r's'\}$ [19]. This shell structure is suited for use of the Dacre scheme [19, 20] to treat molecular symmetry. The integral batch is classified beforehand according to the nature of identities between the four shell labels as shown in Fig. 6. When the inner three loops (DO 110) are completed within a given P shell (DO 100), we have PK integrals P_{pqrs} and K_{pqrs} in which pq runs from $[pmin, 1]$ to $[pmax, pmax]$ and $rs \leq pq$. The inside of DO 120 is the same as in Fig. 5 except a difference that we have to handle PK integrals with several p labels from $pmin$ to $pmax$. This

```
                                        AO members in a shell
      DO   100   P=1, Nshell            ( P : p=pmin, pmax )

      DO   110   Q=1, P                 ( Q : q=qmin, qmax )
      DO   110   R=1, Q                 ( R : r =rmin, rmax )
      DO   110   S=1, R                 ( S : s =smin, smax )

    ( symmetry check for the shell quadruplet {PQRS})

      evaluate one integral batch,
           which is classified according to the nature of identities
           between the four shell labels P, Q, R and S as follows:

    1 PQRS
    2 PQRR & PPRS
    3 PQQS
    4 PQQQ & PPPS
    5 PPQQ
    6 PPPP

      obtain one PK integral batch: P[PQRS] and K[PQRS]

  110 CONTINUE

    ( Here, we have PK integrals P(pq, rs) & K(pq, rs),
           in which pq=[pmin, 1] to [pmax, pmax] & rs.LE.pq)

      DO   120   p=pmin, pmax
      DO   120   q=1, p

      extract Ppq(rs) & Kpq(rs) from P(pq, rs) & K(pq, rs)
          .
          .
    ( the same as in Fig. 5)
          .
          . .
  120 CONTINUE

  100 CONTINUE
```

Fig. 6. Batch-wise PK integral generation

difference requires greater external storage for storing the integrals intermediately, for example, 10 Nao^3 words at maximum when $P = Nshell$ and the P shell is an *spd* shell (totally 10 functions). In order to reduce the external storage requirements, we have only to gather *d*, *pd* and *spd* shells at the lower shell numbers (labels) and *s* shells at the higher ones; work space may be successfully reduced to Nao^3 words at maximum. Furthermore, we should eliminate symmetrically redundant *rs* pairs at the lower *pq* pairs and gather nonredundant integrals at the higher ones in order to achieve enlargement of average vector length *Npq* and *Mpq* of indirect addressing with the index vector *RS*. It is a critical problem in the following steps, Fock matrix generation and MO integral (four-index) transformation.

We now turn to consider which integral form of the three forms, *G* only, *G* and *K* (*GK*) and *PK*, is the best in storing and computing. Table 1 shows relative

Table 1. The relative memory requirement for storing nonzero integrals and their indices in the cases of dense and spatially-extended systems[a]

	Dense systems			Extended systems		
	Integral	Index	Total	Integral	Index	Total
G	3	3	6	1	1	2
P	3	3	6	3	3	6
K	3	3	6	2	2	4
GK	6	3[c]	9	3	3[d]	6
PK[b]	6	3[c]	9	5	3[e]	8

[a] It is supposed that one integral A_{pqrs} ($A = G, P$ or K) and one index pair pq and rs need the same memory unit (e.g. one word length); for example, three integrals (G_{pqrs}; G_{psqr}; G_{prqs}) and three index pairs (pq and rs; ps and qr; pr and qs) are stored for one $pqrs$ combination of G integrals in dense systems, and one integral and one index pair are stored in extended ones

[b] See Figs. 5 and 6

[c] The same index list is used for G and K or for P and K

[d] The different index lists are used for G and K

[e] The index list for K can be included in that for P

memory requirements for storing nonzero integrals and their indices in the cases of dense and spatially-extended systems. In dense systems, three integrals with AO combinations of p, q, r and s are all nonzero (G_{pqrs}, G_{psqr} and G_{prqs}; P_{pqrs}, P_{psqr} and P_{prqs}; K_{pqrs}, K_{psqr} and K_{prqs}). On the other hand, in spatially extended systems [15], only one of the three G integrals (G_{pqrs}) is nonzero when there is large overlap between p and q and between r and s but no (or near zero) overlap between the other AO pairs; therefore, $K_{pqrs} = G_{psqr} + G_{prqs} = 0$. It is needless to say that storing G only is the best in memory requirement and I/O cost. In the Fock matrix generation, we must obtain Coulomb and exchange matrices $\boldsymbol{J}$ and

Table 2. The relative operation number of additions and multiplications needed in obtaining $\boldsymbol{J}$, $\boldsymbol{K}$ and $2\boldsymbol{J} - \boldsymbol{K}$ from G, P and K[a]

	File structure	Dense systems	Extended systems	Vectorizable (O.K.) Unvectorizable (×)
$\boldsymbol{J}$	G	6	2	O.K.
$\boldsymbol{K}$	G[b]	12	4	×
	K	6	4	O.K.
$2\boldsymbol{J} - \boldsymbol{K}$	G[b]	18	6	O.K./×[c]
	GK[d]	12	6	O.K.
	P	6	6	O.K.
$\boldsymbol{J}$ & $\boldsymbol{K}$	G[b]	18	6	O.K./×[c]
	GK[d]	12	6	O.K.
	PK	12	10	O.K.

[a] Operation numbers of index manipulations, e.g. unpacking, are not counted in the table

[b] The usual "purged" form, in which G_{pqrs} is purged when p and q (or r and s) belong to different I.R.s, is not accepted for constructing the $\boldsymbol{K}$ matrix

[c] Vectorizable and unvectorizable for evaluations of $\boldsymbol{J}$ and $\boldsymbol{K}$ matrices, respectively

[d] The G and K integrals are used for evaluations of $\boldsymbol{J}$ and $\boldsymbol{K}$ matrices, respectively

```
      read   Mpq
      read   Ppq, RS
      read   Kpq
      DO   1   n=1, Mpq
      Gpq(n) = Ppq(n) + 0.25*Kpq(n)
        R(n) = indr(RS(n))
1     S(n) = RS(n)-R(n)*(R(n)-1)/2
      IF( p      .EQ.      q  )                          then
      DO   2     n=1,  Mpq
2                              Gpq(n)=Gpq(n)*0.5
                                                         endif
      DO   3     n=1,  Mpq
      IF( R(n)   .EQ.    S(n) )   Gpq(n)=Gpq(n)*0.5
3     IF( pq     .EQ.   RS(n) )   Gpq(n)=Gpq(n)*0.5
    where   indr( r*(r-1)/2+s )=r,      if  r.GE.s ;   R(n).GE.S(n)
```

Fig. 7. How to obtain a canonically-ordered AO integrals G from the PK file

$\boldsymbol{K}$ and, in the closed-shell case, $2\boldsymbol{J}-\boldsymbol{K}$ from one of the three integral files: "less purged" G, "fully purged" GK or "fully purged" PK for the closed-shell and dense systems. Storing of P only is the best in memory requirement, I/O cost and use of the purging technique for Table 2 shows relative operation numbers of additions and multiplications necessary in constructing $\boldsymbol{J}$, $\boldsymbol{K}$ and $2\boldsymbol{J}-\boldsymbol{K}$ from G, P and K. In the closed-shell case, the P integral form is the best even in spatially extended systems. Because the process to obtain $\boldsymbol{K}$ from G needs laborious index manipulations [15] and cannot be vectorized, we had better obtain $\boldsymbol{K}$ from K integrals instead of G. On the other hand, in the MO integral (four-index) transformation, the "full" G integrals are needed; however, the "full" G integrals are obtained from the "full" PK file based on $G_{pqrs} = P_{pqrs} + K_{pqrs}/4$ easily and fast so long as the same index vector RS is used for the P and K integrals as shown in Fig. 7, where the G_{pqrs} integrals are preprocessed when $p = q$, $r = s$ and/or $pq = rs$ [22] and all the loops are automatically vectorized.

5. Vectorization of Fock matrix generation and CI matrix diagonalization

Figure 8 shows the algorithms to generate the "skeleton" closed-shell Fock matrix $\boldsymbol{Fc}$ over AOs from the canonically-ordered list [17] of nonredundant integrals in the form of the AO-based "purged" PK file, where $\boldsymbol{Dc}$ is the one-electron density matrix for the closed shell after doubling of the off-diagonal elements [10]. (The "complete" AO-based Fock matrices can be obtained by filling out the "skeleton" Fock matrices by the symmetrization procedure based on the Dacre scheme [19, 20].) The real symmetric matrix elements of $\boldsymbol{Fc}$ and $\boldsymbol{Dc}$ are packed (linearized) and stored as vectors. The code (8a) is for the use of the original form of P integrals [17]. The code (8b) is for the use of a modified form of P integrals which is often encountered in current SCF codes [10]; that is, P_{pqrs} is beforehand divided by two when $pq = rs$ in order to eliminate its double counting in the Fock matrix generation. For vectorization of DO 110 in Fig. 8, the same Fock matrix

a Fock matrix for the original form of P integrals

```
      DO   100   pq=1, Nao*(Nao+1)/2
      read   Npq
      read   Ppq, RS
      Fcpq      = Fc(pq)
*VOPTION   VEC
      DO   110   n=1, Npq
      Fc(RS(n)) = Fc(RS(n)) + Dc(pq)     * Ppq(n)
      Fcpq      = Fcpq      + Dc(RS(n)) * Ppq(n)
  110 CONTINUE
      Fc(pq)    = Fcpq
  100 CONTINUE
```

b Fock matrix for a modified form of P integrals

```
      DO   100   pq=1, Nao*(Nao+1)/2
      read   Npq
      read   Ppq, RS
      Fcpq      = 0.0
*VOPTION   VEC
      DO   110   n=1, Npq
      Fc(RS(n)) = Fc(RS(n)) + Dc(pq)     *Ppq(n)
      Fcpq      = Fcpq      + Dc(RS(n))*Ppq(n)
  110 CONTINUE
      Fc(pq)=Fc(pq)+Fcpq
  100 CONTINUE
```

Fig. 8. Vectorization of closed-shell Fock matrix generation

elements should not be referenced by $RS(n)$, $n=1$, Npq in Fc ($RS(n)$) at a given pq combination. This property of the index vector RS can be satisfied as described in the preceding section. This guarantee (a special command given by the programmer, for example, *VOPTION VEC in HITAC S-810 and M-680H IAP) is indispensable to forcing the vectorization of the data storing loop by scatter-type indirect addressing with an index vector because any FORTRAN compiler cannot know the contents of the index vector in advance.

In the open-shell case, multiple Fock-like equations are needed. The open-shell SCF calculations are classified as one or partitioned (two, three, . . .) Hamiltonian methods, depending on the number of matrices diagonalized during each iteration [23]. Usually, the Fock-like equations are partitioned and diagonalized sequentially to determine the amount of mixing between only the two shells (among closed, vacant and one or more open shells) at a time; the whole PK integral list is read once for each Fock-like equation. In order to reduce the amount of

disk I/O, however, it may be advantageous to construct as many Fock-like matrices simultaneously per one read of the integral file as available main storage allows; for example, all of the Fock matrices at once, relating to the one Hamiltonian scheme. Most of the one Hamiltonian methods are more difficult in reaching the self-consistent solution than the partitioned Hamiltonian methods and require large MS (main storage) for treating multiple Fock and density matrices. The convergence problem in the one Hamiltonian method has recently been resolved on the basis of partially second-order energy expansions [23-25]. Furthermore, it would seem that the MS requirement of the one Hamiltonian method is not so large in recent high-performance computers. Fig. 9 shows the simultaneous "skeleton" Fock (-like) matrix generation for a system with two open shells, where the density matrix (doubled in the off-diagonal part) and the Fock matrix, ***Dc*** and ***Fc***, are for the closed shell, ***Da***, ***Fa***, ***Db*** and ***Fb*** for the open shells, a and b, and *FACT* is a parameter to specify a multiplet. The MS requirement for all the Fock and density matrices is 3 Nao^2 words; for example, *only* 0.75 MW (mega words) even when $Nao = 500$.

Figure 10 shows the kernel program of the Davidson-Liu method [26, 27] modified by the present author [28] to obtain simultaneously several eigensolutions, either the lowest ones or higher (interior) ones without knowledge of the exact lower ones. This modification attains efficient use of memory space and reduction of iteration cycles, arithmetic operations and I/O processings. In the kernel of the

```
      DO  200  pq=1, Nao*(Nao+1)/2
      read  Npq
      read  Ppq, RS
      read  Kpq
      Fcpq     =Fc(pq)
      Fapq     =Fa(pq)
      Fbpq     =Fb(pq)
*VOPTION  VEC
      DO  210  n=1, Npq
      Kpq(n)   =Kpq(n)*FACT
      Fc(RS(n))=Fc(RS(n))+Dc( pq ) *Ppq(n)
      Fa(RS(n))=Fa(RS(n))+Da( pq ) *Ppq(n)+Db( pq ) *Kpq(n)
      Fb(RS(n))=Fb(RS(n))+Db( pq ) *Ppq(n)+Da( pq ) *Kpq(n)
      Fcpq     =Fcpq     +Dc(RS(n))*Ppq(n)
      Fapq     =Fapq     +Da(RS(n))*Ppq(n)+Db(RS(n))*Kpq(n)
      Fbpq     =Fbpq     +Db(RS(n))*Ppq(n)+Da(RS(n))*Kpq(n)
  210 CONTINUE
      Fc(pq)   =Fcpq
      Fa(pq)   =Fapq
      Fb(pq)   =Fbpq
  200 CONTINUE
```

Fig. 9. Vectorization of open-shell Fock matrix generation

```
      DO  300  P=1, Nci
      read  HP, Q, Np
      IF(NP.GE.NV)          then
      DO  100  i=1, NV
      ZPi       =Z (P, i)
*VOPTION  VEC
      DO  110  n=1, NP
      Z (Q(n), i)=Z (Q(n), i)+C(P,    i)*HP(n)
  110 ZPi       =ZPi        +C(Q(n), i)*HP(n)
      Z (P   , i)=ZPi
  100 CONTINUE
                              else
      DO  201  i=1, NV
  201 ZP(     i)=Z (P   , i)
      DO  210  n=1, NP
      DO  200  i=1, NV
      Z (Q(n), i)=Z (Q(n), i)+C(P,    i)*HP(n)
  200 ZP(     i)=ZP(      i)+C(Q(n), i)*HP(n)
  210 CONTINUE
      DO  202  i=1, NV
  202 Z (P   , i)=ZP(     i)
                                    endif
  300 CONTINUE
```

Fig. 10. Vectorization of CI matrix diagonalization

modified Davidson-Liu method, $\boldsymbol{Z}=\boldsymbol{H}\cdot\boldsymbol{C}$, $\boldsymbol{H}$ is the Hamiltonian matrix of dimension *Nci* and $\boldsymbol{C}_i$'s are the trial vectors for desired solutions (let their number Nexact) and for approximate solutions which need not be obtained exactly (Napprox) and/or the correction vectors (Ncorr). *NV* is the number of vectors (Nexact+Napprox, Ncorr, or Nexact+Napprox+Ncorr) to be treated simultaneously. Only the nonzero Hamiltonian matrix elements H_{PQ} (lower triangle, $P \geq Q$) are stored in external storage in canonical order as P is fixed and Q runs from 1 to P. *NP* is the number of nonzero elements for a specified P. DO 110 of Fig. 10 is basically the same as DO 110 of Fig. 8 in the Fock matrix generation. The code (10) selects the innermost loop which produces the longer vector length *NP* or *NV*. DO 110 uses the index vector Q for indirect addressing of nonzero elements when $NP \geq NV$; otherwise, DO 200 which loops over rows of $\boldsymbol{Z}$ and $\boldsymbol{C}$ is selected. The latter type of addressing causes a serious paging problem in virtual memory systems as in general-purpose computers, but no problem in real-memory systems as in most of supercomputers. We have to pay attention, however, to declaration of the row length of the two-dimensional arrays $\boldsymbol{Z}$ and $\boldsymbol{C}$ so that DO 200 will not encounter memory-bank conflicts in addressing $\boldsymbol{Z}$ or $\boldsymbol{C}$.

6. Vectorization of four-index integral transformation

The present section concerns the four-index integral transformation from the two-electron integrals G_{pqrs}'s over AOs to the integrals V_{ijkl}'s over MOs. We made a comparison between the Bender algorithm [29], which was reviewed in more comprehensive terms by Shavitt [12], and the previously-mentioned (section 2) conventional "two-half" transformation [11], which has been well optimized for vectorization by the Daresbury group (Saunders et al.) [16, 22].

Within the conventional "two-half" transformation formalism (See Sect. 2), Saunders et al. [16, 22] have performed the innermost loops of all the four steps, I, II, III and IV, over MOs, i, j, k and l (their loop lengths always equal the number of MOs to take part in the transformation, Nmo) and employed the Elbert loop structure [11] as the quadruple AO loops (that is, $pq \leq rs$, $p \geq q$ and $r \geq s$). On the other hand, in the Bender algorithm [12, 29], steps I, II and III are computed from all qrs combinations of G_{pqrs} to all ijk combinations of U_{pkji} for a specified p and then, in step IV, V_{lkji}'s are summed up over all $ijkl$ combinations for the p'th AO. The Elbert loop structure over AOs can be incorporated also into the Bender algorithm as shown in Fig. 11. The innermost loop lengths of the modified Bender algorithm are Nmo for steps I and II, $Nmo^2/2$ for step III, and $3Nmo/4$ (average) and $Nmo^2/4$ (average) for step IV; the same for steps I and II as those of Saunders et al. [16, 22] and generally larger for steps III and IV. In the Bender algorithm, the Elbert loop structure *over* MOs is essential in step IV to enlarge the vector length up to $3 \cdot Nmo/4$; otherwise, $3 \cdot Nmo/8$ by the canonical MO loop structure. We have to pay attention to declaration of the row length of the two-dimensional array Kp so that DO 41 will not encounter memory-bank conflicts in addressing of $Kp(ij, l)$.

DO 10 of step I and DO 20 and 21 of step II are the same as the nonzero version of conventional "two-half" transformation by Hegarty [30]. Furthermore, the number of arithmetic operations in all the steps is just the same as that of the conventional transformation algorithms [11, 22] as shown in Table 3, where the ratios of numbers of multiplications in the four steps (and step II′ based on the canonical AO loop structure: $pq \geq rs$, $p \geq q$, $r \geq s$) are compared among various conditions for Nmo, and α denotes the ratio of nonzero AO integrals. Table 3 shows that step IV is the most time-consuming when $Nmo = Nao$ [11, 22], but that in using the canonical AO loop structure step II′ becomes the most expensive if $0.82 > Nmo/Nao > 0.375\ \alpha$ and that in using the Elbert AO loop structure step II and step I the most expensive if $0.58 > Nmo/Nao > 0.75\ \alpha$ and if $Nmo/Nao < 0.75\ \alpha$, respectively; that is, under usual conditions of α (≤ 0.2) and Nmo/Nao (≤ 0.5) for large systems, not step I but step II (or II′) becomes the most time-consuming. Although the Elbert AO loop structure is essential to reduce the number of multiplications by half in the most time-consuming step II, use of the canonical AO loop structure is desirable because we can generate the canonically-ordered PK and G (AO) integrals without laborious sorting procedures, as demonstrated in Sect. 4 (Figs. 5–7). There is one approach, which

```
      DO   100 p=1, Nao
      DO   110 q=1, p
      obtain Gpq, R, S, Mpq
      DO   10  n=1, Mpq
      DO   10  i=1, Nmo
   10 Tpq(i, R(n))=Tpq(i, R(n))+Gpq(n)*X(i, S(n))
      DO   20  r=p, Nao      ( r=1, p  when pq.GE.rs )
      DO   20  i=1, Nmo
      DO   20  j=1, Nmo
   20 Ipq(j, i)=Ipq(j, i)+Tpq(i, r)*X(j, r)
      DO   21  i=1, Nmo
      DO   21  j=1, i
   21 Jpq(ij)=Ipq(i, j)+Ipq(j, i)
      DO   30  k=1, Nmo
      DO   30  ij=1, Nmo*(Nmo+1)/2
   30 Up(ij, k)=Up(ij, k)+Jpq(ij)*X(k, q)
  110 CONTINUE
      DO   40  i=1, Nmo
      DO   40  j=1, i
      read Vij ( ij:Nmo*(Nmo+1)/2 )
      DO   41  k=i, Nmo
      DO   41  l=1, k  ( l=j, k  if  k=i )
   41 Vij(kl)=Vij(kl)+Kp(ij, k)*X(l, p)+Kp(ij, l)*X(k, p)
      DO   42  Kl=ij, Nmo*(Nmo+1)/2
   42 Vij(kl)=Vij(kl)+Kp(kl, i)*X(j, p)+Kp(kl, j)*X(i, p)
      write  Vij
   40 CONTINUE
  100 CONTINUE
```

Fig. 11. Modified Bender algorithm for vectorization

Table 3. Ratio of numbers of multiplications for each step in the four-index transformation[a]

Step	Net cost	Ratio	*Nmo/Nao* 1	0.82	0.63	0.58	0.375	0.75α	0.375α
I	$\alpha\, Nao^4 Nmo/8$	α	α	α	α	α	α	α^*	α^{**}
II	$Nao^3 Nmo^2/6$	$\frac{4}{3}\,(Nmo/Nao)$	1.33	1.09	0.84	0.77*	0.50*	α^*	0.50α
III	$Nao^2 Nmo^3/4$	$2\,(Nmo/Nao)^2$	2	1.33	0.79	0.67	0.28	$1.33\alpha^2$	$0.28\alpha^2$
IV	$Nao\, Nmo^4/2$	$4\,(Nmo/Nao)^3$	4**	2.18**	1*	0.77*	0.21	$1.69\alpha^3$	$0.21\alpha^3$
II′	$Nao^3 Nmo^2/3$	$\frac{8}{3}\,(Nmo/Nao)$	2.67	2.18**	1.68**	1.54**	1**	$2\alpha^{**}$	α^{**}

[a] The inner AO loops over r of step II and step II′ are based on the Elbert loop structure and the canonical loop structure, respectively. ** and * denote steps with the maximum numbers of multiplications in all the steps and in the steps except II′, respectively. In the algorithm (12a), the numbers of multiplications for steps I″ and II″ are $\alpha Nao^4 Nmo/4$ and $Nao^3 Nmo^2/6$, respectively

is available only in the canonical AO loop structure to reduce the number of multiplications of step II′ by half; that is, the number of multiplications is the same as that of step II based on the Elbert AO loop structure. This basic algorithm (step I″ and II″) is shown in Fig. 12a. Although step I″ doubles the number of multiplications of step I, this is not so serious in case of small α and supercomputers to execute vector pipelines in parallel should execute DO 10 of Fig. 12a (step I″) in the same time as DO 10 of Fig. 11 (step I) so long as $R(n) \neq S(n)$. On the other hand, the innermost loop length of step II″ is $Nmo^2/2$, where we should rearrange the index vectors $I(ij)$ and $J(ij)$ so as not to access consecutively to the same memory addresses in $Tpq(I(ij), r)$ and $X(J(ij), r)$ (that is, so as not to encounter memory bank conflicts). It should be noted that, when $\alpha=1$ and $Nmo=Nao$, well-known algorithms based on the canonical loop structure need at least 29 $Nao^5/24$ multiplications [11], while the algorithm (12a) needs 28 $Nao^5/24$ multiplications in spite of using the canonical AO loop structure.

The nonzero AO integrals G_{pqrs}'s for available rs with fixed pq are constructed from the PK file just before being used in step I as discussed in Sect. 4 (Fig. 7). Because the present PK file includes only nonredundant integrals after the Dacre scheme [19, 20] as discussed in the Sects. 4 and 5, symmetrically redundant AO

a Efficient code for the canonical loop (step I″ & II″)

```
      DO  10  n=1, Mpq
      DO  10  i=1, Nmo
      Tpq(i, R(n))=Tpq(i, R(n))+Gpq(n)*X(i, S(n))
   10 Tpq(i, S(n))=Tpq(i, S(n))+Gpq(n)*X(i, R(n))

      DO  20  r=1, p
*VOPTION  VEC
      DO  20  ij=1, Nmo*(Nmo+1)/2
   20 Jpq(IJ(ij))=Jpq(IJ(ij))+Tpq(I(ij), r)*X(J(ij), r)
```

b When symmetrically redundant integrals are reproduced

```
      DO  1   r=1, Nao
    1 blank( r )=.TRUE.
      DO  2   n=1, Mpq
    2 blank(R′(n))=.FALSE.

      DO 10   n=1, Mpq
      DO 10   i=1, Nmo
   10 Tpq(i, R′(n))=Tpq(i, R′(n))+Gpq(n)*X(i, S′(n))

      DO 20   r=1, Nao
      IF(blank(r)) GO TO 20
      DO 21   j=1, Nmo
      DO 21   j=1, Nmo
   21 Ipq(j, i)=Ipq(j, i)+Tpq(i, r)*X(j, r)
   20 CONTINUE
```

Fig. 12. Further modified Bender algorithms

integrals must be recovered by index mapping with the AO correspondence table for the four-index transformation. The algorithm to eliminate a symmetrically redundant integral G'_{pqrs} which is mapped from a nonredundant G_{pqrs} guarantees that $p \geq p'$, $[p, q] \geq [p', q']$ and $[p, q] \geq [r', s']$ but does not guarantee that $[p', q'] \geq [r', s']$. Although the number of multiplications is not affected by such integral recovering, it should be noted that the canonical AO loop structure is not strictly fulfilled; therefore, steps I and II are modified within the present scheme as shown in Fig. 12b. It is most important to reduce the number of *blank* (r) which is .FALSE. in DO 20 of the code (12b), so that it might sometimes be effective to change $R'(n)$ and $S'(n)$ with each other, which is mapped from $R(n)$ and $S(n)$.

We now turn to the consideration of storage requirements of the Bender algorithm [12, 29] and the conventional "two-half" transformation [11, 22]. Main storage requirements of the Bender and the conventional "two-half" transformation algorithms are dominated by $Nmo^3/2$ words for the matrix *Up* and by $Nao^2/2$ words for the matrix *Jij*, respectively. Usually $Nmo > Nao^{2/3}$ ($500^{2/3} = 63$), so that the Bender algorithm requires more MS; when $Nmo = 150$ and 200, $Nmo^3/2$ equals 13 and 31 MB, respectively. Recent large-memory computers can afford to accept these MS requirements. External storage requirements (except the *G* file) of the Bender and the conventional transformation algorithms are for the *V* file and the half-transformed integral *J* file, respectively, where the *J* file ($Nao^2 Nmo^2/4$ words) is always larger than the *V* file ($Nmo^4/8$ words). A clear disadvantage of the conventional transformation is that the I/O processing for the *J* file cannot be eliminated, even if *Nmo* is small enough to keep the whole *V* file in MS. On the other hand, no I/O processing is needed in the Bender algorithm when the whole *V* file can be kept in MS; furthermore, even if the whole *V* file cannot be kept in MS, the numbler of I/O processings of the *V* file can be reduced by half (a fourth, ...) when two (four, ...) matrices with $Nmo^3/2$ words, *Kp*'s, can be kept in MS. An advantage of the Bender algorithm is that the storage requirement, main or external, is only dependent on *Nmo* and the number of I/O processings is easily controllable depending on how many matrices with $Nmo^3/2$ words or how much the *V* file can be kept in MS. Of course, the conventional "two-half" transformation [11, 22] is indispensable when *Nmo* is too large to keep $Nmo^3/2$ words in MS, and is more advantageous when *Nmo* is too large to keep so many matrices with $Nmo^3/2$ words or a greater part of the *V* file and the I/O requirement of the *J* file (at least 2 $Nao^2 Nmo^2/4$) becomes much smaller than that of the *V* file (at most $2 NaoNmo^4/8$). In many cases symmetry blocking can reduce MS requirements, but that is not discussed here, because it has not yet been completely optimized for vectorization.

7. Typical timings and discussion

The present author's SCF-CI program system named GSCF has been under development for about a decade [8], and the current implementation GSCF3 [9] is the result of many revisions to realize and vectorize the "new" conventional SCF-CI algorithm proposed in Sect. 2 (Fig. 2): the reduction of I/O processing

by eliminating sorting routines and redundant integral files and by minimizing the number of iterations in SCF and in CI, the treatment of only nonredundant and nonzero integrals by using vectorizable indirect addressing form with index vectors, and the program structure easily controllable depending on available amounts of MS (main storage) and ES (extended storage). In the AO integral evaluation, GSCF3 is carefully designed to select the most optimal one of the four "extrinsic" vectorization codes (3a), (3b), (4a) and (4b) depending on integral types (Figs. 3 and 4). An appropriate molecular symmetry and its AO correspondence table of the Dacre scheme [19, 20] are automatically generated out of D_{2h} (7 reflection symmetries) and its subgroups without any input data. This accomplished by examining correspondence relations between AOs (Cartesian coordinates, angular quantum numbers, contraction coefficients, orbital exponents) and overlap and one-electron integrals. Although it is very difficult to determine an optimal AO integral storing form among *G*, *GK* and *PK* as was discussed in Sect. 4, the canonically-ordered *PK* integrals are directly generated in GSCF3 (Fig. 6), and used for vectorization of the Fock matrix generation based on the one Hamiltonian method [25] (Figs. 8 and 9), and then internally converted to the *G* integrals (Fig. 7) for the modified Bender algorithm of four-index integral transformation (Figs. 11 and 12). In the SCF calculation, symmetry-adapted initial-guess MOs are automatically generated, without any symmetry specification [31], by using the AO correspondence table and an initial guess of AO occupations. The symmetry-adapted Fock matrix is block-diagonalized within each symmetry block. The Hamiltonian matrix generation program is based on the determinant full-CI expansion [32], but has not yet been completely optimized for vectorization. The CI matrix diagonalization based on the modified Davidson-Liu method [28] is fully vectorized (Fig. 10). The program GSCF3 is registered as a library at the Computer Centre of the University of Tokyo and at the Computer Center of the Institute for Molecular Science. The following timing data were obtained by using the HITAC S-810/model 20 (super) and M-280H and M-680H (general-purpose) installed at the Computer Centre of the University of Tokyo.

Table 4 shows comparison of timings (CPU) among the program systems GAUSSIAN 80 [33], HONDO [34], GSCF2 [8], GSCF3 [9] for the closed shell SCF calculation of ethanol C_2H_5OH with 4-31G* basis functions. The scalar processing unit of the supercomputer HITAC S-810 is just the same as that of the general-purpose computer HITAC M-280H; that is, the S-810 is regarded as an M-280H-based supercomputer. The scalar-adapted version GSCF2 is excellent in speed compared with the GAUSSIAN 80 and the HONDO program systems. The scalar processing time of GSCF3 (*S*-810/scalar) is a little larger than that of GSCF2 (M-280H/scalar), because the vector-adapted version GSCF3 involves overhead in initializing DO loops. This overhead is not so serious, less than 10% in CPU time. The acceleration rates from vectorization on the S-810 are 2.6 in the INT program (1.4 in the Pople-Hehre code, 3.0 in the Gauss-Rys code) and 6.0 in the SCF program; the high-performance vectorization, especially in SCF, can be achieved by GSCF3 even for rather small molecules such as ethanol (only

Table 4. Comparison of CPU times (in seconds) in the integral evaluation (INT) and closed-shell SCF for ethanol 4-31G* calculation (C_s symmetry) by using the programs GAUSSIAN 80, HONDO, GSCF2 and GSCF3

Program	Computer	Mode	INT	SCF[a]	Total
GAUSSIAN 80	M-280H	Scalar	103.5	23.6	127.1
HONDO	M-280H	Scalar	92.7	31.0	123.7
GSCF2	M-280H	Scalar	85.2	14.0	99.2
GSCF3	S-810	Scalar	91.8[b]	14.9	106.7
GSCF3	S-810	Vector	35.9[c]	2.5	38.4
GSCF3	M-680H	Scalar	39.9[d]	6.0	45.9

[a] In all the cases, the number of iterations is 12
[b] It takes 10.6 s and 79.1 s to evaluate sp [13] and spd [14] integrals, respectively
[c] 7.6 s for *sp* and 26.4 s for *spd*
[d] 4.4 s for *sp* and 34.4 s for *spd*

three primitive *d* functions, *Nao* = 57). The HITAC M-680H is the latest general-purpose computer manufactured by Hitachi. It is 2.2 ~ 3.0 and 2.5 ~ 4.0 times faster in scalar and array (IAP) processings, respectively, than the M-280H. A new model of supercomputer based on the HITAC M-680H will be shipped in late 1987. Using the new supercomputer, GSCF3 will be able to complete the 4-31G* ethan calculation within 15 s (INT 14 s, SCF 1 s) in CPU time.

Table 5 shows CPU and elapsed (E) time comparison in the kernel code (8a) of the Fock matrix generation shown in Fig. 8. The *P* integral file of storage amount of 3 MB (4 byte per one integral or one index) is stored in any of external storage (disk), extended storage (ES) and main storage (MS). Each one block (19 *KB* fixed) of the *P* file is read into MS out of disk or ES, and is used in arithmetic operations of the Fock matrix generation. A substantial acceleration in the computation rate by vectorization is attained in the arithmetic operations (~18 times faster, ~115 MFLOPS). The value 115 MFLOPS is rather ideal; suppose the initial setup time for vectorization is not taken into account, and recent supercomputers can perform one gather operation per 12 ~ 15 ns (2 clock cycles)

Table 5. CPU and elapsed time comparison in the kernel code (8a) of the Fock matrix generation (Fig. 8) on the HITAC S-810/model 20 (times in seconds)[a]

(Data storage type)	Disk		*ES*		*MS*	
(Mode)	Scalar	Vector	Scalar	Vector	Scalar	Vector
Arithmetic op. (CPU 1)	0.234	0.013	0.241	0.013	0.238	0.013
Integral read (CPU 2)[b]	0.075	0.075	0.061	0.061	—	—
Total CPU time (1+2)	0.309	0.088	0.302	0.074	0.238	0.013
Elapsed time	3.90	3.94	0.61	0.07	0.35	0.01
Elapsed/CPU	13	45	2	1	1.5	1
Elapsed/CPU 1	17	300	2.5	5	1.5	1

[a] The S-810 is operated in multi-tasking mode with 4 tasks
[b] CPU time required to read *Npq*, *Ppq* and *RS* (total 3 MB data)

and one scatter operation per 24 ~ 30 ns (4 clock cycles). Each one loop operation of DO 110 in Fig. 8 (4 arithmetic operations, 2 Add and 2 Multiply) can be executed theoretically in CPU times of only 24 ~ 30 ns (~150 MFLOPS) because multiple vector pipelines can automatically execute both in parallel and in chained operation mode in recent supercomputers (72 ~ 90 ns per 20 operations, 10 Add and 10 Multiply, as shown in DO 210 of Fig. 9 for the two-open shell case, i.e. ~250 MFLOPS). On the other hand, the CPU speed in reading the *P* integral file is not enhanced and becomes the rate-determining step in the vectorized program. The amount of CPU overhead involved in FORTRAN calls to the I/O service routines is not reduced even if ES is employed instead of disk. The FORTRAN I/O overhead in ES is 20% and 80% of CPU time in scalar and vector processings, respectively. To reduce the FORTRAN I/O overhead, we have to reduce the number of blocks of the *P* file, that is, to enlarge the block size. Needless to say, the best way to avoid the FORTRAN I/O overhead is that the whole integral file is stored in MS [35]. The optimization on GSCF3 using ES is under way in order to obtain a higher transfer speed from ES to MS (up to 1 GB in the S-810/model 20).

Once the CPU time in the SCF calculation has been reduced through vectorization, the overall execution speed is heavily bounded by the I/O efficiency in the Fock matrix generation, as matrix diagonalization is very highly vectorized (for example, it takes only less than 1 s in CPU time to obtain all eigenvalues and eigenvectors of a 300×300 real symmetric matrix), and symmetrization of the "skeleton" Fock matrices and transformation from the AO-based Fock matrices to the MO-based ones are very simple, inexpensive and, of course, highly vectorized procedures. In Table 5, the disk I/O speed is 0.76 MB/s and the CPU requires data transfer speeds of 13 MB/s and 230 MB/s in scalar and vector processings, respectively; the Fock matrix generation causes an exceptionally serious mismatch between the CPU requirement for data supply and the performance of disk systems. Such terrible I/O bottleneck does encourage the use of huge ES or MS to reduce the amount of disk I/O, but otherwise no need of high CPU-performance computers, enhancement of CPU efficiency by vectorization and optimization, nor employment of the canonically-ordered *PK* integral form. On the Hamiltonian matrix diagonalization, it is again desirable to match the CPU requirement for data supply to the performance of disk systems; however, it is less serious because the CPU requirement for supply speed of nonzero Hamiltonian matrix elements H_{PQ} and its index *Q* (see Fig. 10) is ~20 MB/s for a typical value of *NV* (~15) and is satisfied by executing multiple disk I/O channels in parallel (of course, data transfer rate of ES is too enough).

Table 6 shows comparison of CPU times in the SCF-CI calculation for SiF_4. The relative performance of the INT and SCF programs by vectorization is improved over the ethanol calculation, since the SiF_4 system is larger than the 4-31G* ethanol. The performance is improved as more *d* functions and more AOs are used. Furthermore, the programs MOTR and DIAG achieve excellent acceleration rates. The HMAT program has not yet been completely vectorized. Disk storage may be used for the *PK* and *V* integral files because the MOTR and HMAT

Table 6. Comparison of CPU times (in seconds) in conventional SCF-CI calculations for SiF_4[a] by using GSCF3 on the HITAC S-810/model 20

	Scalar	Vector	S/V
INT[b]	280.3	92.9	3.0
SCF[c]	24.3	3.4[g]	7.1
MOTR[d]	714.7	42.1	17.0
HMAT[e]	194.6	92.0	(2.1)
DIAG[f]	336.3	19.3	17.4
Total	1550.2	249.7	6.2

[a] T_d symmetry for MOs, C_{2v} symmetry for integral evaluation based on the Dacre scheme [20]
[b] Basis set: Si[5321/521/1], F[621/51/1], $Nao = 79$
[c] closed-shell SCF, 8 iterations
[d] $Nmo = 28$
[e] $Nci = 6113$
[f] Nexact = 15, Napprox = 0, Ncorr = 15, 13 iterations
[g] The FORTRAN I/O overhead is 2.3 s in CPU time (67%)

programs are CPU-bound. The CPU vs I/O performance in the DIAG program is dependent on the problem, but using disk storage is roughly balanced under a multi-tasking mode with 4 ~ 6 tasks (CPU time is 15 ~ 25%). In the total SCF-CI calculation using GSCF3 on the HITAC *S*-810, the CPU speed in vector mode is 6 times faster than in scalar mode. Table 7 shows CPU time comparison in the SCF calculation for S/Ni_5 as a model for hollow-site adsorption of S on Ni (110) surface. The performance of the INT and SCF programs is still more improved by vectorization. In the SCF step, the FORTRAN I/O overhead is 83% of CPU time and net arithmetic operation time is only 53 s. If the whole integral file is stored in MS, we would achieve a surprising acceleration rate reaching ~60 times (= (3417 − 261)/53). On the other hand, in the SiF_4 calculation, the FORTRAN I/O overhead is 67% and an acceleration rate in the net

Table 7. Comparison of CPU times (in seconds) in conventional SCF calculations for SNi_5[a] by using GSCF3 on the HITAC S-810/model 20

	Scalar	Vector	S/V
INT[b]	9 255	2373	3.9
SCF[c]	3 417	314[d]	10.9
Total	12 672	2687	4.7

[a] C_{2v} symmetry (a model for hollow-site adsorption of S on Ni (110) surface)
[b] Basis set: S[5212111/411111/111], Ni[53321/5311/311]; overall (83*s*/177*p*/168*d*); $Nao = 218$; disk space for the *P* and index file and the *K* file are 425 MB and 199 MB, respectively
[c] Closed-shell SCF, 41 iterations
[d] The FORTRAN I/O overhead is 261 s in CPU time (83%)

arithmetic operation is ~20. In any case, in the GSCF3 program system it is most important to vectorize the AO integral evaluation program more effectively.

8. Conclusions

Experiences with conventional SCF-CI calculations on the HITAC S-810 show that the present strategies for vectorization are satisfactory. As a result of some experiences in rather large-scale SCF-CI computations ($Nao = \sim 200$, $Nmo = \sim 100$, $Nci = \sim 15000$), the vector to scalar acceleration rates are summarized as follows: 2.5~5 in INT, 5~12 in SCF, 15~30 in MOTR, ~2 in HMAT and 10~20 in DIAG, and overall 5~10 through SCF-CI. Although it took ~0.5 man-year to complete modifications and optimizations of the conventional SCF-CI code, the resultant program system has high portability among high-performance computers, super or general-purpose, saving much time and money which can be spent on applications to larger and more complicated systems or more accurate molecular properties. Such achievements are supported by the recent state-of-art technologies of hardware and software. To attain more acceleration in the AO integral evaluation and the Hamiltonian matrix generation programs are the next problems confronting the present author.

Acknowledgements. The author acknowledges the invitation to the Symposium on Computational Chemistry and Parallel Processors held at University of Alberta (June 30-July 2, 1986) by Prof. S. Huzinaga and Dr. M. Klobukowski, and the invaluable information exchange on the use of supercomputers at the symposium. Most of the described modifications on GSCF3 were designed while the author was at Daresbury Laboratory in February 1984. He is grateful to Prof. K. Morokuma for the chance to stay there by the Research Exchange Program between Japan and U.K. and to Dr. V. Saunders for kindly discussing his and Guest's SCF code ATMOL and his and van Lenthe's CI code. The author is indebted to Prof. A. Arima and Dr. Y. Karaki, the late chief director and staff of the Computer Centre of the University of Tokyo, who made great efforts to facilitate large-scale computations, for financial support and priority in using the HITAC S-810/model 20. Finally, the author is deeply thankful to Dr. J. S. Tse for reading the manuscript minutely and correcting the English, the reviewers for many useful comments, and Prof. K. Ruedenberg for arranging this publication.

References

1. Lykos P, Shavitt I (1981) Supercomputers in chemistry, ACS symposium series 173. American Chemical Society, Washington, DC
2. Burke PG, Delves LM (1982) The first international conference on vector and parallel processors in computational science. Comput Phys Comm, vol 26. North-Holland, Amsterdam
3. Dykstra CE (1984) Advanced theories and computational approaches to the electronic structure of molecules, vol 133. NATO ASI series. Reidel, Dordrecht
4. Ahlrichs R, Bohm H-J, Ehrhardt C, Scharf P, Schiffer H, Lischka H, Schindler M (1985) J Comput Chem 6:200
5. Duff IS, Reid JK (1985) The second international conference on vector and parallel processors in computational science. Comput Phys Comm, vol 37. North-Holland, Amsterdam
6. Almlöf J, Faegri Jr K, Korsell K (1982) J Comput Chem 3:385; Almlöf J, Taylor PR, In: [3], p 107
7. Roos BO (1972) Chem Phys Lett 15:153; Roos BO, Siegbahn PEM (1977) The direct configuration interaction method from molecular integrals. In: Schaefer HF (ed) Modern theoretical chemistry, vol. 3. Plenum, New York, p 277

8. Kosugi N (1979) Centre news. Computer Centre, University of Tokyo, 11 [Suppl 2]:115; (1981) ibid, 13 [no 12]:73; (1982) ibid, 14 [no 6]:20; (1982) ibid, 14 [no 7]:52; (1984), ibid, 16 [no 11]:56
9. Kosugi N (1984) Centre news. Computer Centre, University of Tokyo, 16 [no 7&8]:74; (1985) ibid, 17 [no 3]:28; (1985) ibid, 17 [no 9&10]:90; (1985) Supercomputer Workshop Report, vol 4. Computer Center, Institute for Molecular Science, p 109; (1986) GSCF3 Program Manual (SCF part). Computer Centre, University of Tokyo
10. Raffenetti RC (1973) Chem Phys Lett 20:335
11. Elbert ST (1978) Four index integral transformation. In: Moler C, Shavitt I (eds) Numerical algorithms in chemistry: algebraic methods. Lawrence Berkeley Laboratory 8158. University of California, Berkeley, p 129
12. Shavitt I (1977) The method of configuration interaction. In: Schaefer HF (ed) Modern theoretical chemistry, vol 3. Plenum, New York, p 189
13. Pople JA, Hehre WJ (1978) J Comput Phys 27:161
14. Dupuis M, Rys J, King HF (1976) J Chem Phys 65:111; King HF, Dupuis M (1976) J Comput Phys 21:144
15. Guest MF, Wilson S (1981) The use of vector processors in quantum chemistry: experience in the United Kingdom. In: [1], p 1
16. Saunders VR, Guest MF (1982) Applications of the Cray-1 for quantum chemistry calculations. In: [2], p 389
17. Benard M, Speckel B (1982) Comput Chem 6:137
18. Ahlrichs R (1974) Theor Chim Acta 33:157
19. Dupuis M, King HF (1977) Int J Quantum Chem 11:613
20. Dacre PD (1970) Chem Phys Lett 7:47
21. Bair RA (1986) J Comput Chem 7:
22. Saunders VR, van Lenthe JH (1983) Mol Phys 48:923
23. Hsu HL, Davidson ER, Pitzer RM (1976) J Chem Phys 65:609; Davidson ER, Stenkamp LZ (1976) Int J Quantum Chem Symp 10:21
24. Guest MF, Saunders VR (1974) Mol Phys 28:819
25. Kosugi N, Kuroda H (1980) Chem Phys Lett 74:490
26. Davidson ER (1975) J Comput Phys 14:34; Davidson ER (1980) J Phys A13:L179
27. Liu B (1978) The simultaneous expansion method. In: Moler C, Shavitt I (eds) Numerical algorithms in chemistry: algebraic methods. Lawrence Berkeley Laboratory 8158, University of California, Berkeley, p 49
28. Kosugi N (1984) J Comput Phys 55:426
29. Bender CF (1972) J Comput Phys 9:547
30. Hegarty D (1984) Evaluation and processing of integrals. In: [3], p 39
31. Bagus PS, Wahlgren UI (1976) Comput Chem 1:95
32. Kosugi N (1980) On the matrix evaluation in the conventional full configuration interaction. In: Ishiguro E (ed) Contributions from the research group on atoms and molecules, vol. 16. Ochanomizu University, Tokyo, p 10
33. Binkley JS, Whiteside RA, Krishnan R, Seeger R, DeFrees DJ, Schlegel HB, Topiol S, Kahn LR, Pople JA (1980) QCPE 11:406
34. Dupuis M, Sprangler D, Wedolski JJ, NRCC QC01, GAMESS
35. Taylor P (1986) Quantum chemistry calculations using computers with very large central memories, presented at the "Symposium on computational chemistry and parallel processors", Alberta