Theor Chim Acta (1993) 84: 489–509

Theoretica Chimica Acta

A parallel implementation of the COLUMBUS multireference configuration interaction program

Matthias Schüler[1,*], Thomas Kovar[1], Hans Lischka[1], Ron Shepard[2], and Robert J. Harrison[2]

[1] Institut für Theoretische Chemie und Strahlenchemie der Universität Wien, Währinger Strasse 17, A-1090 Wien, Austria

[2] Theoretical Chemistry Group, Chemistry Division, Argonne National Laboratory, Argonne IL 60439, USA

Received February 28, 1992/Accepted May 21, 1992

Summary. In this work a parallel implementation of the COLUMBUS MRSDCI program system is presented. A coarse grain parallelization approach using message passing via the portable toolkit TCGMSG is used. The program is very well portable and runs on shared memory machines like the Cray Y-MP, Alliant FX/2800 or Convex C2 and on distributed memory machines like the iPSC/860. Further implementations on a network of workstations and on the Intel Touchstone Delta are in progress. Overall, results are quite satisfactory considering the complexity and the prodigious requirements, especially the I/O bandwidth, of MRCI programs in general. For our largest test case we obtain a speedup of a factor of 7.2 on an eight processor Cray Y-MP for that section of the program (hamiltonian matrix times trial vector product) which has been parallelized. The speedup for one complete diagonalization iteration amounts to 5.9. An absolute speed close to 1 GFLOPS is found. Results for the iPSC/860 show that ordinary disk I/O is certainly not sufficient in order to guarantee a satisfactory performance. As a solution for that problem, the implementation of a fully asynchronous distributed-memory model for certain data files is in preparation.

Key words: Parallel computing – Multireference CI – COLUMBUS program system

1 Introduction

Parallel computers promise to change the nature of computation in at least two ways. The most dramatic change is that the peak speed of the biggest parallel supercomputers will reach Tera-FLOPS (10^{12} floating point operations per second) performance in the middle of this decade. This speed corresponds to an improvement of four orders of magnitude within two decades taking the Cray 1S of the mid-seventies with 160 MFLOPS peak performance as reference. On the contrary, single processor supercomputer performance will probably not manage

* *On leave from:* Bereich Informatik, Universität Leipzig, Augustusplatz 10/11, O-7010 Leipzig, Germany

a factor of 50 improvement during that same period. Single processor workstation performance – taking the VAX11/780 as standard – is projected to improve by over two orders of magnitude over the same time scale. These TFLOPS machines will only be as expensive as current supercomputers, and will contain many hundreds or thousands of processors. What the programming model will be is still not clear, but what is clear at present is that there will be a two order of magnitude (or more) performance difference between the most powerful parallel and sequential computers.

However, the most widespread impact of parallel computers will probably come in the form of cost-effective few (8–64) processor machines affordable by many small research groups. Such machines might offer increased throughput to a mix of applications, or provide a single modestly parallel resource for a single large application. In contrast to traditional vector supercomputers, these new parallel machines offer much increased scalar performance as well and, indeed, many "non-vectorizable" algorithms are straightforwardly parallelized.

These thoughts are not idle speculation as in many ways that are true already now, albeit on a reduced scale. The fastest supercomputer today is the Intel Touchstone Delta prototype, a parallel computer consisting of a mesh of 528 Intel i860 microprocessors. It has a peak speed of 30 GFLOPS and has been benchmarked at 13.5 GFLOPS (on both the massively parallel LINPACK and the computational kernel of a four-index transformation). Intel and other computer companies, like Cray, are most likely to follow with even more powerful machines. At the moment, and possibly more significant is that many groups are beginning to realize that they already own a very powerful parallel computer in the form of their workstation LAN that is often comparable to a present-day supercomputer in capacity. However, the next realization invariably is that they do not have the software to exploit these resources.

To perform "new science" on these machines we need first to invest the effort required to port our software to this new environment and to develop new methods and algorithms to exploit these new machines. Vector-supercomputers proved themselves well worth such an investment (Refs. [1–5] represent a small collection from among numerous other examples) when coupled with theoretical and algorithmic advances.

Only recently parallel hard- and software have reached a state of reliability and technological standard which makes it worthwhile to attempt the parallelization of such large and complicated programs as *ab initio* electronic structure programs. Parallelization of SCF, CI and Coupled Cluster programs is a field of very active research efforts in many groups (see e.g. [6–10]). However, to our knowledge, so far no successful parallelization of a general MRSDCI program has been reported.

In this paper, we examine in some detail a few-processor parallel implementation of the multireference single- and double-excitation configuration interaction (MRSDCI) program of the COLUMBUS program system [1–3]. Based on our present experience some thought is also given to a massively parallel version of the program.

1.1 The sequential COLUMBUS program

This section is intended to be only a brief characterization of the standard, sequential COLUMBUS Program System. For more information see Refs.

[1–3]. Only those steps which are relevant to our present work of parallelization will be discussed in detail.

The COLUMBUS Program System is a collection of Fortran programs for performing general *ab initio* electronic structure calculations within the framework of MRSDCI. It is based on the Graphical Unitary Group Approach (GUGA) [11, 12] and contains the following program sections:

Atomic orbital (AO)-integral generation, Self-Consistent-Field (SCF), Multiconfiguration SCF (MCSCF), Integral transformation, iterative Davidson diagonalization of a MRSDCI wave function, one- and two-particle density matrices, one-electron expectation and some response properties, MCSCF- and MRSDCI analytical gradients.

The COLUMBUS program runs on a large variety of computers including numerous Unix-based workstations, VAX/VMS minicomputers, IBM mainframes and compatibles, minisupercomputers (including the Alliant FX/8 and FX/2800, Convex C1 and C2, and FPS 500EA), and Cray supercomputers (X-MP, Y-MP, and Cray-2). The individual codes are written and maintained in such a way that porting of the codes to new machines is relatively straightforward. The entire sequential program system (without the gradient part at the time of this writing), including source code, installation scripts, documentation, and sample calculations is available using anonymous FTP from the server ftp.tcg.anl.gov.

The entire electronic structure calculation is performed in a series of steps. The first steps consist of optimization of molecular orbitals (MOs) using the SCF or MCSCF method depending on the case and the complexity of the problem. These MOs are used to define the configuration state functions (CSFs) for the final large-scale CI wave function. This latter wave function is generated from a set of reference CSFs which determine the internal orbital space by allowing single and double excitations into the space of virtual (or external) orbitals. In the approach chosen in the COLUMBUS system always all excitations into the given external space are taken into account thereby allowing the formulation of the Davidson diagonalization (see below) to be broken down into individual dense matrix- and vector-type operations of the dimension of the orbital basis. The structure and numbering scheme of the CSFs is established in the GUGA approach by a distinct row table (DRT) which is constructed in program CIDRT. Next, the internal coupling coefficients for the subsequent diagonalization step are calculated in program CIUFT and stored in a file called the formula tape. After transforming the AO integrals into the MO basis and sorting them in an appropriate way the iterative diagonalization of the matrix representation of the hamiltonian operator is done in program CIUDG. For accurate, large-scale, wave function expansions, this is the computationally most demanding step. MRSDCI expansions of 1–10 million are now becoming routine with the COLUMBUS program system.

Program CIUDG uses the iterative Davidson diagonalization method [13] to determine the appropriate eigenvectors and eigenvalues of the hamiltonian matrix. Most of the effort in large-scale calculations within each of these iterations is the computation of a matrix-vector product of the hamiltonian matrix and a trial vector. These vectors will be called $\boldsymbol{v}$ (trial vector) and $\boldsymbol{w}$ (resulting product vector) in the subsequent sections. The subroutine governing the computation of the product is called MULT. The inherent sparseness of the hamiltonian matrix may be exploited by using the "direct-CI" procedure [14] to compute this matrix-vector product. This involves computing the matrix-vector

product "directly" from the electron repulsion integrals, without explicitly constructing or storing the hamiltonian matrix elements. The advantage of this procedure may be appreciated by noting that for a MRSDCI wave function expansion with orbital basis of size N_{orb}, there are roughly N_{orb}^8 hamiltonian matrix elements total, only $\sim N_{orb}^6$ of which are non-zero, and these non-zero elements are constructed from only $\sim N_{orb}^4$ individual electron repulsion integrals. Since the computational effort scales as N_{orb}^6 while the underlying data scales only as N_{orb}^4, the direct MRSDCI method possesses a quite favorable "surface-to-volume" ratio when comparing the amount of input and output data to the numerical work.

Because of the structure of the MRSDCI wave function as defined above, the overall matrix-vector product involving a very large hamiltonian matrix may be cast into a form [5, 15–17] where most of the operations are of dense matrix-matrix and matrix-vector type where the dimensions of these matrices and vectors are of the length of N_{orb}. In the COLUMBUS program these matrix and vector operations are performed via dense-matrix product kernels (e.g. BLAS(3) routines [18]) which have proved so efficient on vector and scalar-pipelined machines during the past decade. In the coarse-grain parallelization of the COLUMBUS program described in this paper these efficient kernels are still exploited. It is anticipated that this feature will remain quite beneficial also on most of the forseeable parallel computers since the individual nodes can take advantage of these same computational kernels.

During the iterative Davidson procedure the $\boldsymbol{v}$ and $\boldsymbol{w}$ vectors are broken into segments. Subroutine MULT loops over pairs of segments of $\boldsymbol{v}$ and $\boldsymbol{w}$. Thus, only segment pairs need to be kept in central memory, allowing to run calculations with sizes of CI expansions much larger than physical memory. Originally, this feature had been implemented into the sequential code for compatibility reasons in order to make larger CI calculations also possible on machines with rather small central memory. However, as will be demonstrated later, this facility of the program proved to be extremely useful for the purpose of parallelization.

1.2 Objectives

Our objectives separate into short- and long-term goals. In the short term we wish to be able to exploit the resources offered by few processor shared-memory machines (e.g. Alliant FX/2800, Cray Y-MP or Convex C2/C3), networks of workstations, or small configurations of distributed-memory machines such as the Intel iPSC/860. As already mentioned earlier we chose the COLUMBUS program system to start with because it is very well structured and portable to many different machines. In order to achieve our first goal only relatively few structural changes are necessary:

1. A coarse-grain decomposition of the program plus load balancing.

2. Addressing the most gross scaling problems which can be determined from knowledge of the algorithm and empirically without a detailed performance model.

3. Ensuring that I/O communication and memory requirements do not scale excessively with the number of processes.

The long-term goal is to arrive at an MRSDCI program that will scale to massively parallel machines with hundreds of processors. The program that realizes our short-term objectives will have only its most significant bottlenecks

removed and will require only localized modifications of the sequential code. A massively parallel implementation will require construction of a detailed performance model and exploitation of parallelism at all levels of the code, necessitating possibly extensive restructuring and rewriting.

In our present work we concentrated on the most important computational step, i.e. the hamiltonian matrix times trial vector product in program CIUDG of the COLUMBUS program package (see Sect. 1.1). In the meantime a parallel version of the present AO integral program ARGOS [19] has been obtained as well [20]. But it is clear from the beginning that all the other steps mentioned in Sect. 1.1 must be parallelized eventually.

Our decision of how actually to proceed depended on a few other more general considerations which are worth mentioning. First of all, portability is a very important issue. Since the parallelization of the complete COLUMBUS package is a major undertaking it is, of course, extremely desirable that the resulting code operates on as many different machines with a minimum of changes necessary. Since the sequential COLUMBUS program is very well structured and portability is very well taken care of there [3] we had good reasons to expect that we could make good use of these features also here. Software for parallel computers is still expected to undergo basic changes and developments in the future. Thus, we tried to set up our strategy in a way that it is not coupled too strongly to a certain product, and that later changes could be done without too much difficulty. From that point of view (and also for other reasons discussed below) parallelization via compiler is not advisable. Explicit message passing is much more suited for our purpose since it is very simple in its functionality and makes the underlying parallel structures of the programs clear. We do not expect that this is the final way to approach parallelism but, from a pragmatic point of view, this procedure will give us sufficient flexibility for the future with a minimum amount of effort and acceptable performance improvements today.

2 Parallel algorithm

2.1 General considerations

As discussed in the Introduction, the sparse matrix-vector product of the hamiltonian matrix and a trial vector is the most time consuming step for large-scale MRSDCI wave functions. Thus we concentrated in our first efforts to parallelize the COLUMBUS program on that section of the code. This sparse matrix-vector product is needed during the iterative Davidson diagonalization and is part of the program CIUDG. Its overall loop structure is characterized in Fig. 1.

```
loop over pairs of CI vector segments

    loop over types of indices (0-4 internal)

        loop over internal indices

            loop over formulas for a given set of
            internal orbitals

                loop over upper walks

                    dense matrix kernels
```

Fig. 1. Loop structure for the multiplication of the hamiltonian matrix and a trial vector as performed in subroutine MULT( )

The innermost part consists of the dense-matrix kernels mentioned at the end of Sect. 1.1. In a very fundamental way the COLUMBUS CI code and most other modern MRSDCI codes have been exploiting fine grain parallelism via these kernels for many years. Large calculations are reported to sustain over 230 MFLOPS on a single Cray Y-MP cpu [21]. However, the dimension of the matrices involved is usually the number of external orbitals in a symmetry block, which is $O(10-100)$. This is insufficient for efficient distribution on all except the most closely coupled processors (e.g. on 6 processors of an Alliant FX/2800 a 100×100 matrix multiply runs at 156 MFLOPS, a speedup 4.5 relative to the single processor timing). Thus one has to look for coarser granularity at which parallelization should take place.

The DO loops immediately surrounding the low-level matrix operations (see Fig. 1) are over rearrangements of the electrons in the internal orbitals which share the same matrix element structure. In the terminology of GUGA these are the number of upper walks starting from the loop head of an individual loop. Parallelization of this loop would be straightforward as contributions are made to disjoint sections of the result vector. However, there is still insufficient granularity for a distributed memory environment, and worse, the number of upper walks is empirically found to vary greatly, sometimes averaging only 1–3.

Rising further up the nested loop structure we pass through code that is driving the computation for each individual formula of the formula tape, through loops over internal MO indices and over the type of integrals being processed (0–4 internal indices). There is much opportunity for parallelism here, but the structure of the code is quite complex. In addition, the flow of control and data, and the distribution of time, are very sensitive to the nature of the reference space and the DRT.

The outermost loop in the matrix-vector product routine runs over pairs of segments of the trial ($\boldsymbol{v}$) and result ($\boldsymbol{w}$) vectors. This loop is nearly perfect for our purpose.

- It provides the most coarse-grain decomposition possible.
- In a modestly parallel environment we can largely ignore the complexity of the control flow beneath.
- Each process only needs to hold at most four vector segments (two for each $\boldsymbol{v}$ and $\boldsymbol{w}$) and thus the memory requirements do not rise excessively with the number of processors.
- The number of tasks is actually proportional to the square of the number of segments, thus it is possible to have a sufficiently large number of tasks to make load balancing effective.

2.2 Structure of the parallel program

As discussed in Sect. 1.2, we use a message-passing model for parallelization. The actual implementation is performed via the portable programming toolkit TCGMSG developped by one of us (RJH) [22]. The toolkit is available by anonymous FTP from ftp.tcg.anl.gov. TCGMSG supplies a set of Fortran and C callable library routines by which the message passing can be introduced into the application program code. A Single Program Multiple Data (SPMD) approach is used. TCGMSG is available on a large variety of shared memory and distributed-memory machines.

```
MAIN
. . .
   CALL DIAGON()
. . .
END MAIN

SUBROUTINE DIAGON()
. . .
   DO ITER = 1 , NITER    ! Davidson type subspace iteration
. . .
      CALL MULT()
. . .
   ENDDO ITER
END DIAGON

SUBROUTINE MULT()
   DO SEG1 = 1 , NSEG
      READ w_SEG1 , v_SEG1
      DO SEG2 = 1 , SEG1
         READ w_SEG2 , v_SEG2
         UPDATE w_SEG1 , w_SEG2      ! Contributions from H_SEG1,SEG2 and
                                     ! H_SEG2,SEG1 hamiltonian blocks
         WRITE w_SEG2
      ENDDO SEG2
      WRITE w_SEG1
   ENDDO SEG1
END MULT
```

Fig. 2. Structure of the sequential program CIUDG

In Figs. 2 and 3 the basic structures of the sequential and the parallel CIUDG programs are presented. For the present purposes, the most important feature in the sequential code (Fig. 2) are the loops over segment pairs of the v and w vectors. Additional logic (not shown in the figure) is required to handle the SEG1 = SEG2 cases and the case SEG1 = 1, which refers, in our segmentation structure, to CSFs with zero and one external orbital occupancies. In the parallel case (see Fig. 3) all processes start execution at the top of the program. After some initialization and other preparatory work, including setting up a starting guess for the trial vector v, all processes enter subroutine DIAGON( ) where the iterative Davidson diagonalization step is performed. Before calling subroutine MULT( ) (multiplication of the hamiltonian matrix and the trial vectors) process 0 broadcasts a flag telling all other processes to continue with their work. In subroutine MULT( ) we find the identical loop structure over segments as in the sequential case (Fig. 2). However, in addition, the decision is made which process has to update a given segment pair. This is done by a load balancing algorithm. SEG12 effectively maps the double loop over SEG1 and SEG2 into a single loop index. The function nxtval( ) assigns the next free index value from a shared counter to the variable NEXT for each individual process. SEG12 is incremented until the value of NEXT is reached in which case the updating of that particular segment pair is done. Each process owns its local copy of the w vector but shares the trial v vector with the other processes.

When the loops over segment pairs are finished the partial results for the w vector computed by the individual processes are collected by a global sum operation at process 0. In subroutine DIAGON( ) only process 0 is allowed to proceed further and do the remaining computational steps of the Davidson iteration. The other processes remain in a waiting position at the broadcast

```
      MAIN
      . . .
         initialize processes
         NPROC = nnodes() ! total number of processes
         ME = nodeid() ! number of the node
         . . .
         CALL DIAGON()
         . . .
      FLAG = ISTOP
      broadcast FLAG from process 0 to other processes
      CALL pend() ! close process 0
      END MAIN

      SUBROUTINE DIAGON()
      . . .
         DO ITER = 1 , NITER      ! Davidson type subspace iteration
         . . .
            FLAG = IRUN
LOOP:       DO                    ! loop forever
               broadcast FLAG from process 0 to the other processes
               IF (FLAG .EQ. ISTOP) CALL pend() ! all done
               CALL MULT()
               IF (ME .EQ. 0) exit LOOP
            ENDDO LOOP
            . . .                 ! Rest of Davidson iteration
         ENDDO ITER

      END DIAGON

SUBROUTINE MULT()
   SEG12 = 0
   NEXT = nxtval(NPROC)
      ! get index from pool of available indices ( load balancing)
   DO SEG1 = 1 , NSEG
      DO SEG2 = 1 , SEG1
         SEG12 = SEG12 + 1
            IF (NEXT .EQ. SEG12) THEN
            READ v_SEG1 , w_SEG1(ME) ! only if not already in core
            READ v_SEG2 , w_SEG2(ME)
            UPDATE w_SEG1(ME) , w_SEG2(ME)
             ! Contributions from H_SEG1,SEG2 and H_SEG2,SEG1 hamiltonian blocks
            WRITE w_SEG2(ME) to local file
            get NEXT from pool of available indices
                                          ! load balancing
         ENDIF
      ENDDO SEG2
      if update done on segment SEG1 WRITE w_SEG1(ME) to local
      file
   ENDDO SEG1
   global sum of w to processor 0
END MULT
```

Fig. 3. Structure of the parallel program CIUDG

statement ready to enter MULT( ) for the next time. In case that the Davidson iterations are not finished yet process 0 broadcasts a message to the other processes to continue (FLAG = IRUN). Otherwise process 0 leaves subroutine DIAGON( ) and FLAG is set to ISTOP. Broadcasting FLAG now by process 0 causes all other processes to quit. Finally process 0 is closed as well.

As has already been mentioned above, presently, several files are assumed shared between processes. File organization and access mode in subroutine

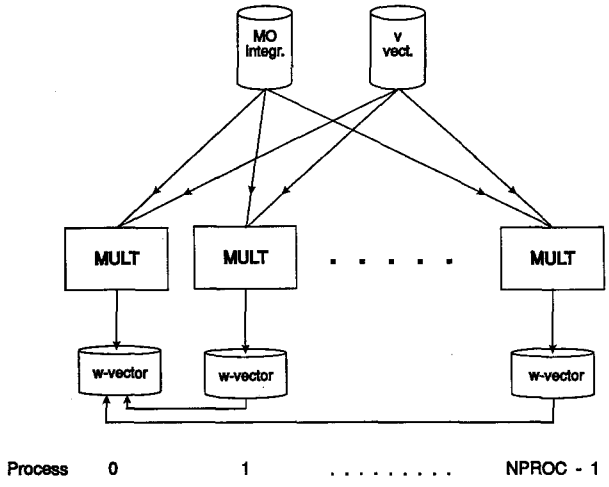

Fig. 4. Data flow in subroutine MULT( )

MULT( ) is shown in Fig. 4. During construction of the matrix-vector product, segments of the CI vector are read by all processes from a single shared file. This file is updated by process zero during the Davidson diagonalization procedure. In order to ensure that all write buffers are flushed and that read buffers are invalidated it has been found necessary to close and re-open this file in all processes immediately prior to the global barrier implied by the broadcast of FLAG (in subroutine MULT( )) described above.

The implementation of the procedures just described by subroutine calls to the TCGMSG library is straightforward. Basically, the modifications in subroutine MULT( ) were the only ones worthwhile mentioning with one important exception. In the original sequential program the segment structure of the CI vector is determined for a given amount of core memory so as to *minimize* the number of segments. In the sequential case, there is no advantage to achieve a balance of the size of different segments. Typically, the maximum number of segments is 3 to 5; in most applications enough memory is available to hold the complete v and w vectors in core. In the case of the parallel program the situation is different. A larger number of segments is required (typically 20 to 50) in order to balance the work load over processes. Unfortunately, the sequential program showed in such cases (for which it was not designed) an approximately quadratic increase of computer time with the number of segments (see Fig. 5 and Sect. 3.1). By analyzing the sequential program code we found the reason for this increase in the processing of the formula tape. Since only one formula tape containing the information for all CSFs was constructed, for each segment pair, the complete file had to be read, the information for each formula had to be unpacked and checked whether it contributed to the particular segment pair. Thus, especially in multireference cases for which the formula tape is very long, the amount of effort to extract useful information for one segment pair was very small compared to the total amount of work of analyzing the entire formula tape. There were two ways out of that dilemma. One was to sort and split the formula tape into several files. This procedure had the disadvantage that the structure of the formula tape depended on the segmentation of the CI vector and had to be changed with changing segmentation. The amount of I/O for reading the formula tape would still have remained appreciable. The solution adopted

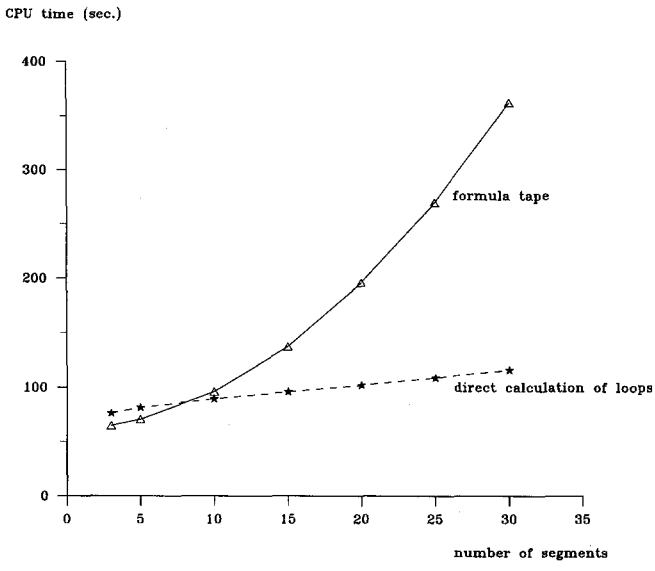

Fig. 5. Comparison of timings (for one complete CI iteration, Cray Y-MP) for the formula tape case vs. direct calculation of the internal contribution to the CI matrix elements. The C_{2v}-*pVTZ* test case was used

was to abandon the formula tape completely and to recalculate, for each segment pair, specifically the necessary formulas on the fly. In that way the respective I/O was eliminated completely at the cost of the cpu time required to calculate the formulas directly. This cost for the formula generation only depends on the structure of the internal space. Except for overlapping cases between segment boundaries, work can be organized so that each formula is calculated only once. Thus, the computational work connected with the formula calculation would be more or less independent of the segmentation scheme. However, since some overhead with setting up the DO loops in the DRT and validating the internal walks on the DRT was necessary we obtained linear dependence on the number of segments. This is a great progress compared to the previous quadratic behavior. Detailed timings will be given in Sect. 3.1.

3 Performance

The parallel CIUDG program is currently running on a rather wide selection of machines: Sun workstations, Personal Iris, Convex C2, Cray Y-MP, Alliant FX/2800 and Intel iPSC/860. Adaption of the program to other machines, like the IBM RS/6000 series or the Intel Touchstone Delta will follow in the near future. In this section we present timings for three typical test cases in order to demonstrate the efficiency but also the bottlenecks of our program (for an evaluation of these timings see also authors' note at the end of the article).

These test examples were set up in the following way: all calculations were performed for the electronic ground state of the CH_3 radical. A CAS (complete active space) with 7 valence orbitals was chosen for the determination of the reference configurations. Within that configuration space MCSCF calculations were performed. For the CI expansion all configurations belonging to the CAS (without application of symmetry restrictions) were used to construct all single and double excitations from the valence orbitals into the full virtual orbital

space. The K-shell orbital was frozen. For more details on the calculations see Ref. [23]. Depending on the basis set and the geometry 3 different cases were finally treated:

(a) C_{2v}-*pVDZ*: The molecular geometry was D_{3h} symmetry ($R_{CH} = 2.039$ bohr). Only C_{2v} symmetry was utilized in our program. The basis set was of polarized-valence double-zeta (*cc-pVDZ*) quality and taken from the compilation by Dunning [24]. It consisted of a $9s4p1d$ Gaussian basis on carbon contracted to [3,2,1] using general contraction techniques. For hydrogen a $4s1p$ basis contracted to [2,1] was used. The 29 orbitals were distributed over the four irreducible representations of C_{2v} as a_1 14, b_1 5, b_2 8 and a_2 2. The CAS consisted of 2–5 a_1, 1–2 b_2 and 1 b_1 orbitals which gave 188 reference configurations. The dimension of the final CI expansion was 70,254.

(b) C_{2v}-*pVTZ*: This case is identical to the previous one except that the AO basis was of polarized-valence triple-zeta (*cc-pVTZ*) quality [24] (C: $10s5p2d1f$ [4,3,2,1], H: $5s2p1d$ [3,2,1]). The 72 basis functions were distributed over the irreducible representations as: a_1 30, b_1 14, b_2 20, a_2 8. The size of the CI expansion was 624,334 CSFs.

(c) C_1-*pVTZ*: A distorted geometry of C_1 symmetry and the same *pVTZ* basis as in case (b) was chosen. The same CAS (7 active valence space orbitals) as above gave rise to 784 reference configurations which resulted in a CI expansion of 2,528,400 CSF.

3.1 Efficiency of the segmentation scheme

The dependence of computer time on the segmentation of the $\boldsymbol{v}$ and $\boldsymbol{w}$ vectors is of crucial importance since the distribution of work to the individual processor is done via segment pairs, and load balancing requires that the size of the segments is not too large. Typically, we used 20 to 50 segments. Ideally, the total execution time should remain unaffected by the segmentation. That this was not the case for our original sequential program has already been discussed in Sect. 2.2.

In addition, the I/O requirements in connection with the segmentation scheme have to be considered as well. As discussed previously, the multiplication $\boldsymbol{H} \cdot \boldsymbol{v}$ is driven by the four indices of the two-electron integrals. There are five cases which are classified according to the number of external indices as 4-, 3-, 2-, 1- and 0-external. In the 4-, 3- and 1-external cases the updating of $\boldsymbol{w}$ can be achieved by passing once through all segments. In the remaining 2- and 0-external cases each segment pair has to be considered individually. Thus, the 4-, 3- and 1-external integrals have to be read from disk $N_{\text{seg}}/2$ times and the 2- and 0-external integrals $N_{\text{seg}}(N_{\text{seg}} - 1)/2$ times where N_{seg} is the number of segments. The factor of one half for the first part of integrals comes from the fact that in this case the passage through the segment pairs can be organized in a way that two consecutive segments are combined thus reducing the reading of the 4-, 3- and 1-external integrals by approximately a factor of two. Moreover, the $\boldsymbol{v}$ vector segments have to be read from disk and the $\boldsymbol{w}$ vector segments have to be read from and written to disk. The amount of data to be transferred in that case is $3/2$ $(N_{\text{seg}} - 1)N_{\text{CI}}$ in working precision units.

The efficiency of the segmentation scheme depends on other factors as well. The direct calculation of the formula tape information is basically scalar whereas

Table 1. Timings for the sequential CIUDG program in dependence on the segmentation of the CI vector[a,b]

	4-ext.	3-ext.	2-ext.	1-ext.	0-ext.	$H \cdot v$	Total
Cray Y-MP							
C_{2v}-*pVDZ*							
no segmentation	0.77	1.9	13.0	16.2	4.4	36.2	36.3
30 segments	0.77	2.1	27.4	16.7	23.4	70.6	70.7
C_{2v}-*pVTZ*							
no segmentation	10.3	9.1	32.4	18.4	4.8	75.0 (75.3)	75.3 (75.9)
30 segments	10.3	9.4	52.1	18.8	23.4	114.7 (117.6)	115.1 (118.2)
C_1-*pVTZ*							
no segmentation	93.9	46.9	229.5	72.6	25.7	468.6 (469.2)	470.0 (471.6)
30 segments	94.0	47.8	265.2	73.1	87.6	569.8 (580.6)	571.2 (583.1)
Alliant FX/2800							
C_{2v}-*pVDZ*							
no segmentation	6.7	20.9	69.1	56.5	10.9	164.6	166.7
30 segments	6.9	21.3	132.1	57.7	46.4	271.4	273.6
C_{2v}-*pVTZ*							
no segmentation	367.1	354.4	568.5	139.1	52.7	1485.3	1508.2
30 segments	374.9	357.7	904.6	140.2	88.1	1909.0	1932.2
Convex C220							
C_{2v}-*pVDZ*							
no segmentation	4.9	12.4	49.0	52.4	16.0	134.7	135.3
30 segments	4.9	12.9	127.0	54.2	71.8	273.1	273.8
C_{2v}-*pVTZ*							
no segmentation	136.6	127.4	234.0	88.1	31.2	617.6	623.7
30 segments	137.4	130.3	452.3	89.5	88.9	904.4	910.5

[a] Timings in cpu seconds for one Davidson iteration (wall clock times are given in parentheses)
[b] For the definition of the individual contributions see the text

for the dense matrix operations vector operations can be used. Also, I/O is accounted for differently on different machines. Thus, in order to give an overview over the performance of the segmentation we consider it useful to discuss timings on a series of computers. The influence of all these individual factors can already be investigated at a single processor level. Thus, for the timings described in this section we used for simplicity our sequential program extended by the possibility of direct calculation of the formula tape information. Since the program code for updating one segment pair of w is identical in the sequential and the parallel program the conclusion drawn here fully pertains to the calculations with the parallel program to be described in the next section.

In Table 1 timings for calculations with and without segmentation of v and w are compared for several computers. In the table the total time for one Davidson iteration is split up into its individual components according to the classification of the two-electron integral indices as explained above. As to be expected, segmentation has by far the largest effect for the 2- and 0-external cases which depend – as explained above – on contributions from each pair of segments.

In case of the Cray Y-MP these times were further broken down by means of Profiling [25] (see Table 2). In this table the total time for one iteration is

Table 2. Individual timings on the Cray Y-MP for the sequential CIUDG program[a,b]

	C_{2v}-$pVDZ$		C_{2v}-$pVTZ$		C_1-$pVTZ$	
	no segm.	30 segm.	no segm.	30 segm.	no segm.	30 segm.
formula calc.						
3-ext.	0.1	0.1	0.1	0.1	0.1	0.1
2-ext.	2.5	13.0	2.3	11.6	3.1	13.5
1-ext.	9.3	9.8	9.2	9.7	9.9	9.6
0-ext.	4.2	21.5	4.0	21.4	23.5	80.0
formula tp. (total)	16.1 (39%)	44.4 (60%)	15.6 (19%)	42.8 (35%)	36.6 (7%)	103.2 (17%)
matr. multipl. plus overhead	25.7 (61%)	29.7 (40%)	64.8 (81%)	78.4 (65%)	479.5 (93%)	501.4 (83%)
total	41.8	74.1	80.4	121.2	516.1	604.6

[a] Timings in cpu seconds for one Davidson iteration
[b] For the definition of the individual contributions see the text

broken down into contributions due to the formula calculation, the dense matrix routines plus overhead including logic and cpu time assigned to the I/O. The charging of cpu time for I/O is almost negligible here. The three examples are arranged in increasing work done in the external space. This is achieved by either increasing the basis set ($DZ \rightarrow TZ$) or by reducing the symmetry ($C_{2v} \rightarrow C_1$). The generation of the formula data is independent of the external orbital space. This fact is nicely reflected in the timings of Table 2. The timings for the 3- and 1-external cases are practically independent of segmentation whereas in the 2- and 0-external cases an increase in computer time can be observed. However, with one exception, timings remain constant for a given segmentation which means that with increasing basis size (keeping the internal space unchanged) the relative importance of the formula computation is significantly reduced. Only in the case of the 0-external (all-internal) indices the time for the formula calculation increases when going from C_{2v} to C_1 symmetry because this is the only case where it is possible to take symmetry into account. The formula calculation amounts to 60% of the total time for one iteration in the C_{2v}-$pVDZ$ case. However, this example has been included for tesing purposes only and contains a rather small external orbital space. It is not characteristic for high-level calculations. The proportion of the formula calculation is significantly reduced for the larger basis sets. It could have been still further reduced if the work for the external space had been increased by increasing the basis set and not by reducing the symmetry. Figure 5 shows a comparison of timings for the old program where the complete formula tape was read for each segment pair with those obtained with the present scheme. The advantages of our new procedure are obvious. One can see that in this latter case computer time increases linear with N_{seg}. This is an acceptable behavior since the number of tasks which can be produced in that way increases quadratically with N_{seg}. One should also mention here that the CAS reference space is a very demanding one and that for more restricted MR cases the importance of the formula generation will decrease significantly. The timings for the unsegmented calculations show that the recalculation of the formulas is also an interesting alternative for the sequential

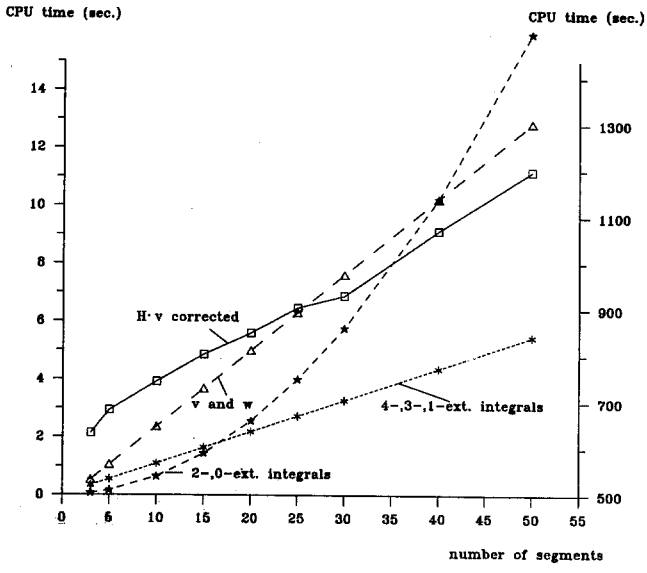

Fig. 6. Timings for reading and writing the two-electron integrals and the v and w vectors (left coordinate axis) and for $H \cdot v$ reduced by the cpu time attributed to I/O (right coordinate axis). The data were obtained on a Convex C2 and the C_{2v}-$pVTZ$ test case was used

program itself. The break-even point between the versions of reading the complete formula (old) and recalculating only what is needed (new) is situated in all cases investigated at rather small segment numbers (6 to 7).

The timings for Convex and Alliant (see Table 1) show a dependence on segmentation which is very similar to those for the Cray. Again, the 2- and 0-external cases show a significant increase in computer time from the unsegmented to the 30 segment case. Contrary to the Cray data transfer from and to disk is also reflected in the cpu timings. With the help of a simple test program the cpu time per MB transferred data was determined and the actual cpu times were calculated from the known amount of data transferred in each iteration cycle. In Fig. 6 the relative importance of individual contributions is depicted for the Convex C2. Timings for the 4-, 3- and 1-external integrals and the v and w vectors depend linearily on the number of segments. The 2- and 0-external integral file is much smaller than the 4-, 3- and 1-external file. However, since this file is read $N_{\text{seg}}(N_{\text{seg}}-1)/2$ times (as opposed to $N_{\text{seg}}/2$ times for the other part of the integral file) it soon becomes a prominent factor and finally dominates the I/O. Data transfer for the v and w vectors turns out to be a very important factor as well. The cpu time for the hamiltonian matrix times vector product (excluding the cpu time for the I/O) behaves almost linearily with increasing number of segments. A similar situation is also found in case of the Alliant.

3.2 The parallel program

After having investigated the characteristic behavior of the sequential program in some detail we are now ready to discuss the performance of the parallel program. We concentrate on two questions: how well does our load balancing scheme work and what speedup can actually be achieved? An unambiguous answer can only be obtained on dedicated machines without any other jobs interfering. In

the following, we will discuss such results obtained on an eight processor Cray Y-MP and an Alliant FX/2800 by comparing cpu and wall clock times. For the Convex C2 for which a dedicated system was not available to us at present. In addition to the just mentioned shared memory machines, we also will present results for the iPSC/860.

3.2.1 Cray Y-MP. Table 3 shows timings (cpu- and wall clock times) obtained on a dedicated eight processor Cray Y-MP for our two larger test cases. The results are overall very satisfactory. Comparison of cpu and wall clock times (no given in the table) for the individual steps of $\boldsymbol{H} \cdot \boldsymbol{v}$ shows very small time differences which means that I/O is very well taken care of (standard Fortran I/O routines have been used throughout except for the 4- and 3-external two-electron integral files for which asynchronous AQREAD [26] routines were employed).

Table 3. Timings for the parallel CIUDG program on a dedicated 8 processor Cray Y-MP[a,b]

	Proc. no.	4-ext.	3-ext.	2-ext.	1-ext.	0-ext.	$\boldsymbol{H} \cdot \boldsymbol{v}$		Total	
		cpu	cpu	cpu	cpu	cpu	cpu	wall cl.	cpu	wall cl.
C_{2v}-*pVTZ*										
1 proc.	0	10.4	9.4	51.9	18.9	23.1	114.5	118.4	114.9	119.2
4 procs.	0	3.6	3.6	11.4	5.7	4.8	29.3	30.5	29.7	32.6
	1	1.5	1.1	17.2	2.0	6.9	29.0	30.5		
	2	3.2	2.7	9.5	8.2	5.1	28.9	30.5		
	3	2.1	2.0	13.7	3.0	6.3	27.3	30.5		
8 procs.	0	1.4	1.9	4.8	2.8	2.1	13.1	16.6	13.6	20.5
	1	0.7	0.5	7.8	0.8	3.4	13.4	16.6		
	2	1.5	1.1	3.5	5.7	2.7	14.6	16.6		
	3	0.7	0.6	8.5	0.9	3.7	14.5	16.6		
	4	1.8	1.6	6.1	2.6	2.5	14.6	16.6		
	5	1.5	1.3	7.4	2.3	3.1	15.7	16.5		
	6	1.4	1.1	6.9	1.8	2.5	13.8	16.6		
	7	1.5	1.3	7.3	2.1	3.2	15.5	16.5		
C_1-*pVTZ*										
1 proc.	0	94.1	47.7	264.8	73.3	86.4	568.4	578.8	570.0	581.9
4 procs.	0	22.8	10.4	74.8	15.7	18.2	142.5	149.9	144.2	158.3
	1	26.1	16.5	59.5	22.7	20.0	145.4	149.9		
	2	19.4	9.5	71.5	13.9	20.6	135.5	149.9		
	3	26.3	11.6	59.6	21.3	27.8	147.1	149.9		
8 procs.	0	13.3	7.7	18.2	14.8	14.0	68.2	80.9	70.0	98.0
	1	13.1	5.3	34.3	8.0	9.6	70.6	80.9		
	2	16.4	8.3	28.1	11.6	8.1	72.8	80.9		
	3	6.5	2.8	44.2	4.1	13.7	71.8	80.9		
	4	6.6	3.5	37.7	5.2	11.6	65.0	80.9		
	5	13.1	8.8	26.2	11.6	8.8	68.9	80.9		
	6	13.0	4.4	40.8	7.4	11.4	77.4	80.9		
	7	13.1	7.6	36.6	11.3	9.8	78.8	80.9		

[a] Timings in cpu and wall clock seconds for one Davidson iteration with 30 segments
[b] For the definition of the individual contributions see the text

The total timings for the $\boldsymbol{H} \cdot \boldsymbol{v}$ step are an indication of how well our load balancing procedure works. The cpu times are a measure of the work done by each process. The wall clock times are identical for all processes because of the necessary synchronization after finishing the $\boldsymbol{H} \cdot \boldsymbol{v}$ calculation. The average cpu efficiency as defined as average cpu time for $\boldsymbol{H} \cdot \boldsymbol{v}$ divided by the wall clock time is 97%, 94% and 87% for the 1, 4 and 8 process calculations in the C_{2v}-*pVTZ* case and 98%, 95% and 89% in the C_1-*pVTZ* case. The total timings (last two columns in Table 3) include the global sum for the $\boldsymbol{w}$ vector and the Davidson subspace manipulation which is done by process zero only. The cpu time for this step constitutes only a minor amount whereas the wall clock time increases significantly with the number of processors. This is possibly due to a non-optimal installation of the global sum operation. The speedups for the $\boldsymbol{H} \cdot \boldsymbol{v}$ step (wall clock time for the single-processor calculation divided by the respective wall clock time for the multiple-processor calculation) are:

C_{2v}-*pVTZ* 3.9 (4 processes) and 7.1 (8 processes)

and

C_1-*pVTZ* 3.9 (4 processes) and 7.2 (8 processes).

The speedups based on timings for one complete iteration are reduced to 3.7/5.8 (C_{2v}-*pVTZ*) and 3.7/5.9 (C_1-*pVTZ*) because of the deterioration of the performance as just mentioned. MFLOPS rates have been determined by the hardware processor monitor (hpm [25]). Since in our test runs on the dedicated computer we had performed only one full iteration the overhead from initial (B_k iteration [27]) and final (Davidson correction [28]) steps deteriorate the results in an unbalanced way. Therefore we tried to correct for these influences and estimated the true MFLOPS rate in the C_1-*pVTZ* case as follows: for a single process on the dedicated computer one Davidson iteration including the aforementioned overhead gave 144 MFLOPS per cpu second and 138 MFLOP per wall clock second. Since we obtain a speedup of a factor of 7.2 for the $\boldsymbol{H} \cdot \boldsymbol{v}$ step in case of eight processes we have achieved about 990 MFLOPS for that part of the program. Since the overall increase for the complete iteration is (for the moment) only a factor of 5.9 the overall speed is 814 MFLOPS. In calculating relative speedups we have taken as reference so far a sequential calculation with segmentation. The numbers obtained in this way give a realistic picture of the efficiency with which the individual processors are utilized. However, from the point of view of throughput one has of course to compare with the most efficient sequential case which is the unsegmented one. If one takes the wall clock times for the unsegmented calculation as given in Table 1 the speedup for the 8 processor case (C_1-*pVTZ*) is reduced to 4.8.

3.2.2 Alliant FX/2800 and Convex C2. Results for Alliant and Convex are presented in Table 4. For the Alliant a dedicated computer was available. Thus, cpu and wall clock times are given. Test runs on the Convex could only be performed under heavy load. Therefore, we tabulate for Convex cpu times only.

The distribution of cpu times over the individual processes shows again that load balancing works quite well. However, in case of the Alliant we encountered one special problem. For all other machines investigated the sum of the cpu times for a given segmentation was practically independent of the number of processes. As Table 4 shows this is not the case for the Alliant FX/2800. The total cpu time (sum over all processes) for $\boldsymbol{H} \cdot \boldsymbol{v}$ increases from 268.8 sec (1 process) via 279.3 sec (4 processes) to 390.1 sec (8 processes). Thus, even though

Table 4. Timings for the parallel CIUDG program on a dedicated Alliant FX/2800 and a Convex C220[a,b]

	Proc. no.	4-ext.	3-ext.	2-ext.	1-ext.	0-ext.	$H \cdot v$		Total	
		cpu	cpu	cpu	cpu	cpu	cpu	wall cl.	cpu	wall cl.
Alliant FX/2800										
C_{2v}-*pVDZ*										
(30 segm.)										
1 proc.	0	5.3	17.4	129.1	61.1	46.9	268.8	271.0	271.8	274.0
4 procs.	0	1.2	3.9	39.6	12.2	13.2	75.6	77.0	79.9	81.0
	1	2.2	7.5	28.5	22.4	8.9	74.0	77.0		
	2	0.8	2.2	41.8	6.8	14.6	71.9	77.0		
	3	1.6	4.8	28.4	23.8	12.4	75.8	77.0		
8 procs.	0	0.5	2.3	23.8	6.6	6.9	46.6	52.0	52.6	60.0
	1	0.5	1.7	25.7	5.0	7.8	47.1	52.0		
	2	0.6	1.5	25.3	4.2	8.5	46.5	52.0		
	3	1.4	4.8	18.6	14.1	5.0	49.7	52.0		
	4	0.9	4.3	22.7	11.2	6.7	51.6	52.0		
	5	0.9	2.1	24.1	7.4	7.1	47.8	52.0		
	6	1.4	4.8	19.0	14.7	5.8	51.5	52.0		
	7	0.7	1.9	17.8	15.9	8.5	49.3	52.0		
Convex C220										
C_{2v}-*pVTZ*										
1 proc.	0	146.5	138.1	458.6	88.2	83.9	921.4		928.3	
4 procs.	0	34.8	29.4	126.3	16.6	20.8	229.9			
	1	30.8	31.9	120.6	16.6	22.4	224.1			
	2	31.3	25.7	129.4	26.6	25.9	240.6			
	3	51.5	53.5	89.0	29.6	15.3	240.2			

[a] Timings in cpu and wall clock seconds for one Davidson iteration with 30 segments
[b] For the definition of the individual contributions see the text

parallelism is very well established the final speedups are poor because of the increase of the total cpu time. From the behavior of CIUDG on all other computers we were sure that this increase in cpu time could not come from our code directly. Tests with simple benchmark programs finally showed that the I/O performed in parallel was responsible for that artificial increase of the cpu time. Such a behavior was not observed on other systems. Since this problem is clearly related to the operating system we do not see any possibility at the moment to improve the situation.

3.2.3 Intel iPSC/860. In contrast to the previously described shared memory machines the iPSC is a distributed-memory machine. However, these architectural differences are hidden by TCGMSG. In the way we had designed the parallel CIUDG program it was possible to port CIUDG to the iPSC without changing a single line of code concerning parallelization. What had to be changed was linking the appropriate library for BLAS routines and routines concerning I/O which is managed via the Concurrent File System [29]. Only wall clock times are available. As TCGMSG works at the moment one node is dedicated to the nextval( ) server which is responsible for load balancing. This

Table 5. Timings for the parallel CIUDG program on the iPSC/860[a,b,c]

	Proc. no.	4-ext.	3-ext.	2-ext.	1-ext.	0-ext.	Sum	$H \cdot v$[d]	dgop[e]	Total
1 processor	0	8.3	12.9	182.7	66.4	74.8	345.1	403.1	5.9	415.6
3 processors	0	3.6	5.2	53.4	33.8	25.3	121.3	146.4	6.7	159.5
	1	3.1	4.2	66.9	18.1	24.5	116.8	146.4	1.3	
	2	2.7	3.6	66.0	14.8	26.2	113.3	146.4	1.5	
sum		9.4	13.0	186.3	66.7	76.0	351.4			
7 processors	0	2.1	2.1	29.0	20.0	16.3	69.5	98.3	16.3	121.1
	1	1.9	1.6	34.4	6.5	14.0	58.4	98.1	5.1	
	2	2.2	2.1	32.3	9.0	13.4	59.0	98.3	5.2	
	3	0.9	0.8	33.4	3.4	14.7	53.2	98.1	4.6	
	4	1.0	1.1	37.3	4.5	14.0	57.9	98.1	5.1	
	5	2.6	3.1	31.9	12.3	12.9	62.8	98.0	5.0	
	6	2.6	2.6	30.5	11.4	12.0	59.1	98.1	4.7	
sum		13.3	13.4	228.8	67.1	97.3	419.9			

[a] Timings in wall clock seconds for one Davidson iteration
[b] For the definition of the individual contributions see the text
[c] The C_{2v}-*pVDZ* test case with 30 segments is used
[d] Includes I/O for reading and writing the v and w vectors but excludes the time for the global sum
[e] Global sum (see Fig. 3)

node, even though formally used during the execution of CIUDG, is not counted in the following discussion.

First tests very quickly showed that efficient treatment of I/O is crucial for working with the iPSC. For the 4- and 3-external integral files we use the asynchronous IREAD( ) routine [29] and for the other integral files and for reading and writing the v and w segments CREAD( ) and CWRITE( ) [29]. The results of our final calculations are collected in Table 5. Looking first at the timings of the individual sections 4- to 0-external and the sum for each process (horizontal sum) one finds that load balancing works about as well as for our tests on the other computers. The increase in total time spent for each case (vertical sums) with the number of processors is attributed to the decreasing efficiency to access the integral files individually. This increase is especially remarkable for the 2- and 0-external cases. The difference between the timings in the horizontal sums and those in column $H \cdot v$ are due to reading and writing the v and w vectors. The effort necessary for these disk operations is still substantial. The just described trends continue when the number of processors is increased. Calculations with 15 processors do not show any additional speedup compared to the 7 processor case.

It would be possible to improve the I/O for the 2-, 1- and 0-external integrals by introducing asynchronous I/O in the same way as was done for the 4- and 3-external integrals. Performing the I/O for the v and w vectors is more difficult since these files are treated in direct access mode. In any case, it seems to be clear that all these further improvements of disk I/O would not lead to satisfactory results. We intend to go into a completely different direction and to remove the disk I/O completely by moving towards a fully asynchronous distributed-memory model as described in Sect. 4.

4 Conclusions and outlook

We could show that our coarse-grain approach to parallelize the CI section of the COLUMBUS program is a successful one. On a 8 processor Cray Y-MP we have achieved a speed close to 1 GFLOP per wall clock second for the most important part of the program. The changes which are necessary to port the program from one parallel computer to the other are minimal and are automated. However, we also have made it clear that, because of the complexity and requirements of MRCI programs in general, the work described here has to be regarded as a first step. Within the present structure a few technical improvements, including the acceleration of the direct formula calculation and of the global sum and data compression schemes [30] for the v and w vectors must be made. With these changes a working version of CIUDG for routine applications will be available for shared memory machines. It is also evident from our results for the iPSC that these developments alone will not be sufficient. We will have to do more about data structures.

One way to treat structures more efficiently is to move away from the excessively synchronous message-passing model (some synchronization will always be necessary since we do not have unlimited buffer space available) to a fully asynchronous distributed-memory model. Such a model, closely related to Linda [31], is described in Ref. [32]. Basically, buffers of data are put into a globally accessible space which is uniformly distributed across all processes. An event-driven mechanism provides fully asynchronous access for all processes to all of the data, with a bandwidth for distributed accesses that scales up with the number of processes. Work in this direction is in progress on the Intel iPSC/860 hypercube and is also expected to be of similar benefit on shared-memory parallel computers such as the Cray Y-MP or the Alliant FX/2800.

This distributed-memory model can be used to eliminate the excess I/O on the 2-external integrals. The I/O on the v and w vector files scales approximately linearly with the number of segments, and since each process has its own local copy the total disk space increases linearly with the number of processes. It is proposed to buffer this data with a distributed data model also, exploiting the coordination properties of the Linda-like primitives [31, 32] to handle the mutual exclusion required for correct updating. An alternative solution is to use a single shared file with locks at the segment level.

One of the biggest requirements for disk space usually arises from the 3- and 4-external integrals. For very large systems this is a bottleneck in even the sequential code and work is underway to change to an AO driven scheme [33–36]. Initially, the AO integrals will be stored on disk, employing sparsity. However, this then readies us for a simple transformation to a so-called "double-direct" MRSDCI algorithm, recomputing the integrals only as needed. The 0-, 1- and 2-external integrals are still processed in the MO basis.

At this point we fully expect to have a program that performs effectively on few (10–30) processor computers, depending somewhat on the nature of the computation and how effective the I/O subsystem is. The just described program features will also put us into the position to effectively use workstations within local area networks which, as has already been stressed in the beginning, represent a very valuable and widespread source for computational power.

To go beyond this level, to a massively parallel version, we need to eliminate essentially anything that is either serial or actually increases with the number of processes/segments. Almost all I/O must be eliminated and the Davidson

algorithm must be modified [37] so that only one full v and w vector need be manipulated. Amdahl's law [38] is a severe task master on massively parallel machines. To get a speedup of just 500 on a 1000 processor machine (only 50% efficiency) the combination of the serial work and overhead due to parallelization or lack of load balancing must be less than 0.1% of the parallel work. Such a decomposition would give rise to an efficiency of 99.2% on just 10 processors. An efficiency of just 50% seems low if performance is our only goal. However, it still should be enough that calculations with $O(10^8)$ or more configurations in very large MO basis sets could become quite routine on large parallel computers.

Authors' note

The COLUMBUS program system timings presented in this paper represent preliminary work and the individual codes have been optimized to different extents on the various computer systems. Consequently, these timings should not be used directly to assess either the ultimate performance of the COLUMBUS program system or to compare the performance of the different computer systems.

Acknowledgements. This work was performed under the auspices of the Office of Basic Energy Sciences, Division of Chemical Sciences, U.S. Department of Energy, contract number W-31-109-Eng-38, and the Austrian "Fonds zur Förderung der wissenschaftlichen Forschung", Project numbers P7174 and P7979. We are also grateful for support by the "Österreichische Forschungsgemeinschaft", Project number 06/1454, and by the Commission of the European Communities (Codest), Contract No. SC1*915086. We thank Cray Research for supplying us with generous support and ample computer time at a Cray Y-MP in Eagan, Minnesota. The calculations on the iPSC/860 were performed at the ACRF of the Argonne National Laboratory and at the CCSF at CalTech, those on the Alliant FX/2800 at the facilities of the Theoretical Chemistry Group, Chemistry Division, Argonne National Laboratory and those on the Convex C220 at the EDV Zentrum of the University Innsbruck. We also are grateful to the Pacific Northwest Laboratory (PNL) for access to the Concurrent Supercomputer Consortium iPSC/860 at CalTech and to Rick Kendall from PNL for helpful discussions.

References

1. Lischka H, Shepard R, Brown F, Shavitt I (1981) Intern J Quantum Chem S15:91
2. Ahlrichs R, Böhm H-J, Erhardt C, Scharf P, Schiffer H, Lischka H, Schindler M (1985) J Comp Chem 6:200
3. Shepard R, Shavitt I, Pitzer RM, Comeau DC, Pepper M, Lischka H, Szalay PG, Ahlrichs R, Brown FB, Zhao JG (1988) Intern J Quantum Chem S22:149
4. Saunders VR, Guest MF (1982) Comput Phys Commun 26:389
5. Saunders VR, van Lenthe JH (1983) Mol Phys 48:923
6. Whiteside RA, Binkley J St, Colvin ME, Schaefer III HF (1987) J Chem Phys 86:2185
7. Colvin ME, Whiteside RA, Schaefer III HF (1989) In: Wilson S (ed) Methods in computational chemistry, vol 3. Plenum, NY
8. Luethi HP, Almlöf J (1992) University of Minnesota Supercomputer Institute Research Report UMSI 91/249, Minneapolis, Minnesota
9. Rendell AP, Lee TJ, Lindh R (1991) Daresbury Laboratory, Daresbury, Warrington
10. Brode S, Ahlrichs R, A distributed implementation of TURBOMOLE(DSCF), private communication
11. Paldus J (1981) In: Hinze J (ed) The unitary group for the evaluation of electronic energy matrix elements. Springer-Verlag, Berlin, p 1

12. (a) Shavitt I (1981) In: Hinze J (ed) The unitary group for the evaluation of electronic energy matrix elements. Springer-Verlag, Berlin, p 51
(b) Shavitt I (1988) In: Truhlar DG (ed) Mathematical frontiers in computational and chemical physics. Springer-Verlag, Berlin, p 300
13. Davidson ER (1975) J Comput Phys 17:87
14. (a) Roos BO (1972) Chem Phys Lett 15:153
(b) Roos BO, Siegbahn PEM (1977) In: Schaefer III HF (ed) Methods of electronic structure theory. Plenum, NY, p 277
15. Werner HJ, Reinsch EA (1982) J Chem Phys 76:3144 and references therein
16. Ahlrichs R (1983) In: Diercksen GHF, Wilson S (eds) Methods in computational molecular physics. Reidel, Dordrecht, p 209
17. Meyer W, Ahlrichs R, Dykstra CE (1984) In: Dykstra CE (ed), Vectorization of advanced methods for molecular electronic structure. NATO ASI, Reidel, Dordrecht
18. (a) Dongarra JJ, DuCroz J, Hammarling S, Hanson R (1988) ACM Trans on Math Soft 14:1
(b) Dongarra JJ, DuCroz J, Duff I, Hammarling S (1990) ACM Trans on Math Soft 16:1
19. Pitzer RM, Shepard R (1983) In: Annual Report of the Theoretical Chemistry Group, October 1982 to September 1983, Argonne National Laboratory, Argonne IL, USA
20. Harrison RJ, Kendall RA (1991) Theor Chim Acta 79:337
21. Shepard R (1990), unpublished results, presented at the 45th Ohio State University Symposium on Molecular Sepctroscopy, Columbus, Ohio
22. Harrison RJ (1991) Intern J Quant Chem 40:847
23. Aoyagi M, Shepard R, Wagner AF (1991) Intern J Supercomputer Applications
24. Dunning Jr. TH (1989) J Chem Phys 90:1007
25. UNICOS Performance Utilities (1991) Reference Manual SR-2040. Cray Research Inc., Mendota Heights, MN 55120, USA
26. UNICOS Fortran Library Reference Manual SR-2079. Cray Research Inc., Mendota Heights, MN 55120, USA
27. Shavitt I (1977) The method of configuration interaction. In: Shcaefer III HF (ed) Methods of electronic structure theory. Plenum, NY, p 189
28. (a) Davidson ER (1974) In: Daudel R, Pullman B (eds) The world of quantum chemistry. Reidel, Dordrecht, p 17
(b) Langhoff SR, Davidson ER (1975) Int J Quantum Chem S9:183
29. iPSC/2 and iPSC/860 Programmer's Reference Manual (1990). Intel Scientific Computers, Beaverton, Oregon
30. Shepard R (1990) J Comp Chem 11:45
31. (a) Carriero N, Gelernter D (1989) Comm of the ACM 32:444
(b) Carriero N, Gelernter D (1990) How to write parallel programs. A first course. The MIT Press, Cambridge, MA
32. Harrison RJ, Theor Chim Acta (submitted for publication)
33. Meyer W (1977) Configuration expansions by means of pseudonatural orbitals. In: Schaefer III HF (ed) Methods of electronic structure theory. Plenum, NY, p 413
34. (a) Werner HJ, Reinsch EA (1981) In: Van Duijnen TH, Niewpoort WC (eds) Proc 5th Seminar on Computational Methods in Quantum Chemistry. MPI Garching München
(b) Werner HJ, Reinsch EA (1982) J Chem Phys 76:3144
35. Ahlrichs R (1981) In: Van Duijnen TH, Niewpoort WC (eds) Proc 5th Seminar on Computational Methods in Quantum Chemistry. MPI Garching, München
36. Kovar T, Lischka H, work in progress
37. Davidson ER (1989) Comp Phys Comm 53:49
38. Amdahl GM (1967) Proc AFIPS Spring Joint Computer Conf 30:40