Theor Chim Acta (1994) 89: 77–88

Theoretica Chimica Acta

Parallel algorithm for integral transformations and GUGA MCSCF

Theresa L. Windus, Michael W. Schmidt, Mark S. Gordon
Department of Chemistry, Iowa State University, Ames, IA 50011, USA

Received December 22, 1993/Accepted March 4, 1994

Summary. An algorithm for the parallelization of the atomic to molecular integral transformation and the subsequent steps in a GUGA based MCSCF calculation is presented. Timing data shows that the transformation and diagonalization steps are well parallelized and that several of the other portions of the MCSCF code are moderately parallel. Remaining sequential bottlenecks are identified.

Key words: Parallel MCSCF – Integral transformation

1 Introduction

Over the last seven years, several papers have appeared that present algorithms for the parallelization of the self-consistent field (SCF) portions of *ab initio* electronic structure codes [1, 2]. However, parallel implementation of correlated (post Hartree-Fock) methods have only recently been explored. Second-order Moller-Plesset perturbation theory (MP2) [3], coupled cluster [4], and multi-reference configuration interaction (MRCI) [5] methods have all been examined for parallel content.

A computational bottleneck common to all post Hartree-Fock methods is the transformation of the integrals in the atomic orbital (AO) basis into the molecular orbital (MO) basis. Several stand alone parallel transformation programs have been considered [6], but these were not incorporated into a practical application. Watts and Dupuis have presented a parallel transformation algorithm for shared memory machines which was used for parallel MP2 calculations [3].

In this paper, we describe a parallel transformation algorithm and the application of this algorithm to the parallelization of a MCSCF program based on a graphical unitary group approach (GUGA) CI program. This parallel algorithm has been implemented into the General Atomic and Molecular Electronic Structure System (GAMESS) [2] quantum chemistry code.

2 Parallel algorithm and implementation

In the full optimized reaction space (FORS) [7] and complete active space SCF (CASSCF) [8] approach to MCSCF calculations, one partitions the molecular

orbitals into several subspaces. These include the core orbitals (occupied with a fixed occupancy of two electrons), partially occupied orbitals, and empty virtual orbitals. The fractionally occupied valence orbitals are referred to as the "active space", and all possible configurations involving the active electrons are used. Once these spaces have been chosen by the user, the MCSCF wavefunction is completely specified. A good overview of MCSCF algorithms has been given by Roos [9].

The steps which must be performed by GAMESS in order to obtain an MCSCF energy are shown schematically in Fig. 1. The basic steps are: 1) Obtain an initial guess for the molecular orbitals (MOs) (these are usually obtained from an

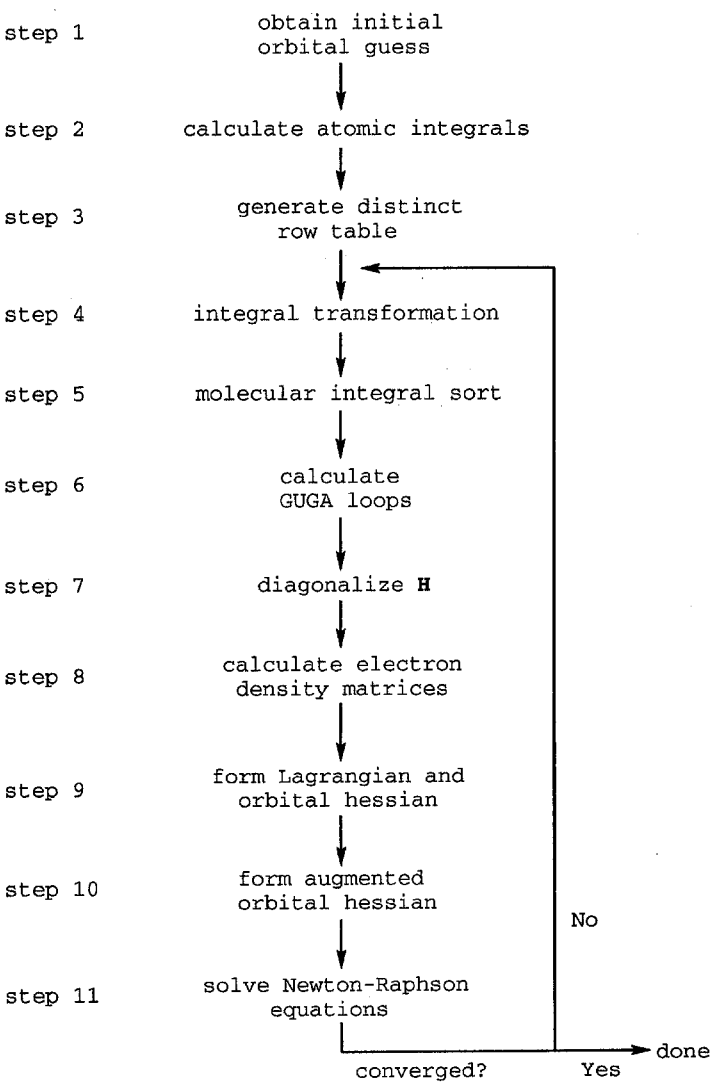

Fig. 1. Schematic of the GUGA MCSCF steps

SCF calculation, but do not have to be). 2) Calculate the integrals in the atomic orbital (AO) basis. 3) Generate the distinct row table which contains information describing the configuration state functions (CSFs) used in the MCSCF wavefunction. 4) Transform the AO integrals into the current MO basis. 5) Sort only those transformed integrals with all indices in the core and active spaces into an order needed by step 6. 6) Calculate contributions (known as GUGA loops) to the CI Hamiltonian matrix, $\boldsymbol{H}$. 7) Diagonalize $\boldsymbol{H}$ to obtain the CI eigenvectors, $\boldsymbol{C}$. 8) Calculate the 1 and 2 electron density matrices. 9) Form the orbital gradient (Lagrangian) and orbital hessian. 10) Form the augmented orbital hessian. 11) Solve the Newton–Raphson equations to improve the orbitals. 12) Repeat steps 4 through 11 until the desired convergence is achieved.

In our algorithm, we try to parallelize as many of the steps in Fig. 1 as possible without *major* revisions of the code.

The single-program multi-data (SPMD) model has been adopted for GAMESS [2]. In this model each node performs all of the tasks (the "peer" model), executes essentially the same code, and needs to have most (if not all) of the information (basis set, geometry, orbitals, etc.) necessary for the calculation. The major exception to the peer model is that only one node (the "master") reads the input and writes the output. The master broadcasts information from the input file to all other nodes. The majority of this broadcasting is performed in the initial setup part of the calculation.

The SPMD model has been very useful in that it is relatively easy for a new section of code to be run in parallel. In fact, the current algorithm requires only about thirty lines of parallel code (not including code that was initially modified for the SCF).

The message passing library TCGMSG [10] developed by Robert Harrison is used to perform the necessary communication. TCGMSG uses the best communication available for the particular architecture being used. It works on distributed memory machines (such as the Intel Delta), shared memory machines (such as an Alliant), and an Ethernetwork of UNIX workstations (possibly with different platforms). This portability is the main reason we chose TCGMSG to perform the communications.

We use two different methods to perform load balancing. One is a static model (loop) and the other is a dynamic model (nxtval). The loop model essentially assigns every nth loop in a DO loop to the nth processor. No communication is required in this type of load balancing and it generally works well when each loop has about the same amount of work and/or there are many loops. This method also works best on processors of equal speed and work load.

The nxtval method uses a shared counter to distribute the work. TCGMSG keeps track of the counter (generally on the master node) and increments it each time a processor calls for more work. This, of course, requires communication. Therefore, this model works best when the amount of work in each loop is fairly large (i.e. the communication is a small fraction of the compute time). This is the model that we use when the processors are not of the same speed and/or work load.

Before discussing the parallel steps needed for each iteration of the MCSCF energy, the first three steps of the energy calculation must be discussed. In step 1, the initial orbital guess, the starting orbitals are usually obtained from an SCF calculation or a previous MCSCF calculation. If the orbitals are read from an input stream, the master reads and broadcasts them to the other nodes. This step is only performed once in the calculation and does not require much time (as will be shown in the timing examples).

Step 2, calculation of integrals in the AO basis, is also only performed once in the energy calculation. In our previous implementation of parallel SCF [2] this step was shown to be highly parallel. However, for the MCSCF implementation, it is necessary for all nodes to have a complete list of atomic integrals available (the reason for this will be explained in the integral transformation discussion below) and therefore, the AO integrals are calculated sequentially on each node. Since this sequential step is performed only once and is approximately of order N^4 (where N is the number of basis functions), the overall cost is minimal compared to the repeated N^5 integral transformations. If there is not enough disk space available on each node to hold the atomic integrals, they are recalculated each time they are needed in the integral transformation step (in a direct method).

Generation of the distinct row table (DRT) defining the CSFs [11], step 3, is also performed just once in the energy calculation. Each node builds the table and stores all of the information into a local disk file. This information is needed throughout the rest of the calculation; therefore, it is necessary for each node to have the complete DRT. An alternative would be to let only the master node build the table and then broadcast the DRT data when it is needed. However, to avoid as much communication as possible during the iterative steps, we chose to allow each node to calculate and store this information.

The first three steps discussed above are sequential, but they can be considered to be setup for the MCSCF energy iterations, since they are only performed once. The next steps are executed during each MCSCF iteration and have been parallelized as much as possible.

A schematic of the parallel algorithm for the integral transformation (step 4) is given in Fig. 2. We have incorporated a new transformation algorithm from HONDO [12] into our code. This transformation can use an unsorted list of AO integrals and makes use of molecular symmetry (Abelian groups only) [13]. A key feature of this algorithm is that it calculates only those transformed integrals that are needed in the MCSCF. These are the $\langle ij \mid kl \rangle$, $\langle aj \mid kl \rangle$, $\langle ab \mid kl \rangle$, $\langle aj \mid lb \rangle$ and $\langle aj \mid bl \rangle$ integrals, where i, j, k, l are MOs in the core and active spaces and a, b are MOs in the virtual space.

One option for performing the transformation consists of passes over the complete list of atomic integrals, where each pass generates a subset of the required molecular integrals. A pass consists of generating all ij index pairs for a given subset of kl. Since each node has a complete list of atomic integrals, this option is perfect for parallelization. The number of passes is chosen to be as evenly divisible by the number of processors as possible, ensuring load balance. Unfortunately, as will be seen in the first example, it is not always possible to perfectly divide the number of passes by the number of processors.

Each node ends up with only a subset of the transformed integrals on its disk. The beauty of this algorithm is that there is no communication involved (unless nxtval balancing is used) and it is very easy to implement (as shown in Fig. 2). Note that only the master node transforms the one-electron integrals.

The next task is to make each of the subsequent MCSCF steps work with the distributed molecular integral list and to parallelize where possible. Step 5 (sorting of the transformed integrals) is generally performed in memory since no integrals involving virtual MOs are needed and thus, memory for this step is usually available. First, the array that holds the sorted integrals is zeroed ensuring a correct global sum of the sort array. Then, each node sorts its subset of the integrals into the appropriate position in the sort array. After all of the nodes have completed the sort, a global sum is performed on the sorted array. Thus, all nodes

```
      SUBROUTINE TRANSFORM
C
C     ME = the processor's ID number
C     NPROC = number of processors
C
C     initialize parallel
C
      IPCOUNT = ME -1
      NEXT = -1
      MINE = -1
C
C     Code here to determine number of passes.
C

C
C     Begin passes over the atomic integrals to form subsets
C     of the transformed integrals (IJ/KL).
C
      MINKL = 1
   10 MAXKL = MINKL + SIZE_OF_PASS
C
C     Select nxtval or loop balancing ...
C
         IF (PARALLEL) THEN
            IF (NXTVAL) THEN
               MINE = MINE + 1
               IF (MINE.GT.NEXT) NEXT = NXTVAL()
               IF (NEXT.NE.MINE) GO TO 20
            ELSE
               IPCOUNT = IPCOUNT + 1
               IF (MOD(IPCOUNT,NPROC).NE.0) GO TO 20
            END IF
         END IF
         CALL SUBTRANS()
   20    CONTINUE
         IF (MAXKL.EQ.NUMKL) THEN GO TO 30
C
C     NUMKL = maximum number of KL indices
C
         MINKL = MAXKL + 1
         GO TO 10
   30 CONTINUE
      END
```

Fig. 2. Schematic of parallel integral transformation

end up with the identical, complete sorted list of occupied MO integrals which they store onto disk.

The global sum of the sorted integrals is essential for the next step (6). The sorted integrals are used to calculate contributions (GUGA loops [11]) to the CI Hamiltonian matrix, $\boldsymbol{H}$. Since several integrals can contribute to the same loop, it is indeed convenient for all of the nodes to have all of the necessary integrals.

The GUGA loop generation part of the code is not easy to parallelize. This part of the code can use a large amount of CPU time (see examples below), but more importantly it can also require considerable disk space to store the generated loops. Ideally, we would like to make sure that 1) each node performs only the calculations needed to evaluate its assigned loops (i.e. distribute the CPU time evenly across the nodes) and 2) each node ends up with about the same number of loops stored on disk (i.e. distribute the loop storage evenly across the nodes). Unfortunately, many parts of the existing loop generation code have data dependencies

that we have been unable to avoid (i.e. lines of code where $x(n)=x(n-1)+y$). Parallelization at a higher level in the subroutine results in very poor load balance in both time and disk space. Therefore, we have opted to put the parallel calls around the subroutine that actually completes each loop's computation and writes it to disk. This means that the disk space needed to store loops is very evenly distributed, but that there is only limited savings (or parallel content) in the CPU time needed to calculate the GUGA loops. The decision to evenly divide the loops over the nodes, rather than to minimize CPU, is driven by the parallel requirements of the subsequent diagonalization step. The even distribution of the disk space also permits problems that would not fit on one node because of disk space limitations to be run on several nodes where the combined disk space is enough to hold all of the loops. Additional improvement in the parallel nature of this part of the code will probably require a total rewrite of the loop generation algorithm to evenly distribute the computational time.

The next step is to diagonalize $\boldsymbol{H}$ to obtain the CI eigenvectors [14]. This step, 7, involves reading in the loops from disk to form $\boldsymbol{HC}$, where $\boldsymbol{C}$ is the current approximation to the CI eigenvector. Formation of this matrix vector product is by far the most time consuming step in an iterative Davidson diagonalization procedure. Each iteration requires an exact matrix diagonalization in an iterative subspace (of dimension less than 30, typically). This and other steps in the Davidson diagonalization are negligible compared to the formation of $\boldsymbol{HC}$, and are therefore run sequentially.

Since the loops are distributed across the nodes, each node evaluates its partial contribution to $\boldsymbol{HC}$ and then a global sum is performed. Because the loops were evenly distributed across the nodes by the previous step, this part of the step is essentially perfectly parallel.

During the CI diagonalization setup, loops contributing to the diagonal elements are processed to form the total contribution to the diagonal of $\boldsymbol{H}$. Unit vectors corresponding to the lowest diagonal elements are generated as the initial approximation to each desired eigenvector, $\boldsymbol{C}$. Diagonal elements of $\boldsymbol{H}$ are then retained in memory to avoid the need to process the diagonal loops again. Their contribution to $\boldsymbol{HC}$ during the Davidson iterations is made correct by scaling each diagonal element by $1/p$, p = number of processors. The final global sum of $\boldsymbol{HC}$ over p processors thus includes diagonal terms only once. Step 7 actually shows very nice speed-ups as will be demonstrated below.

Upon finishing the CI diagonalization step, each node has the CI vectors and eigenvalues. The CI vectors are then used to generate the 1 and 2 electron density matrices (step 8). This step works in a manner similar to that of step 6. The CI vectors are used to generate loops which contribute to the density matrices [15]. The loops are actually generated by the same subroutine that generates the loops in step 6 and the comments on load balance from that discussion also apply here. The main parallelization of the loops is to distribute the disk space needed for the loops over all of the nodes. The largest difference between this step and step 6 is that the number of loops needed to generate the density is generally less than the number of loops needed to generate $\boldsymbol{H}$. After the loops are generated, they are sorted into a form needed by step 9. Since each node has only a subset of the loops, a global sum of the density is performed after the sorting step. This sort and global summation is similar to that in step 5, and results in each node having the complete density.

The next step (9) is the formation of the orbital gradient (Lagrangian asymmetry) and orbital hessian (hessian) [16]. The Lagrangian and hessian are formed

by combining the transformed integrals with elements of the density matrix. This is generally the part of the MCSCF calculation that requires the greatest amount of memory (to simultaneously hold the hessian and two-electron density in memory). In our current implementation, each node needs as much memory as it would need to run sequentially. Each node reads the complete density matrix into memory from disk. Then, each node reads buffer loads of its partial list of transformed integrals and computes its partial contribution to the Lagrangian and hessian.

As mentioned earlier, only the master node has the one-electron transformed integrals, so that node evaluates the one-electron contributions to the Lagrangian and hessian. After all nodes have completed their contributions, global sums are performed on the Lagrangian and hessian, resulting in each node having the complete Lagrangian and hessian.

The next two steps, formation of the augmented orbital hessian [17], which adds the orbital gradient to the hessian to improve numerical stability, and solution of the Newton–Raphson (NR) equations for improving the orbitals, are sequential steps. Solution of the NR equations amounts to finding the lowest eigenvector of the augmented hessian, and is currently performed sequentially by each node. Timing results presented in the next section reveal that this is an obvious place for future improvements in the parallel algorithm.

Once the orbital improvements are made, convergence is checked. If the wavefunction (and energy) is not converged, steps 4 through 11 are then repeated.

3 Timing examples

One of the interesting aspects of MCSCF is that different types of calculations cause different parts of the code to be the primary bottleneck. For example, a calculation with many core orbitals and a relatively small active space will have the integral transformation and NR orbital improvement as its bottleneck. On the other hand, a calculation with only a few occupied orbitals but a relatively large active space will have its bottleneck at the CI diagonalization step. To illustrate this large variation in the computational bottleneck, three different types of examples are used for the timing tests. These examples are indicative of the range of MCSCF calculations performed with GAMESS.

All of the tests were performed on RS/6000 model 350s connected by an Ethernet. The machines were dedicated for these tests, but the network was not isolated. Therefore, other packets on the network could interfere with the TCGMSG communications. Since all of the nodes are of the same speed and load, only the loop (static) balancing was used in these examples. Also, all of the examples use C_1 symmetry. Even though we are able to use symmetry in each step of the MCSCF energy calculation, C_1 symmetry is used to provide a fair comparison between the different types of test examples and to make one iteration long enough for timing information to be meaningful.

The first example is the twisted transition state of H_2CNH using a triple-zeta valence (TZV) [18] basis set (TZV++(2df, 2pd)) totaling 128 basis functions. There are 6 core orbitals and the active space consists of four electrons in four orbitals (one doubly occupied, one alpha occupied, one beta occupied, and one unoccupied in the reference) which generates only 20 configuration state functions (CSFs). 1.5 MW of memory and approximately 350 MB of disk are needed. The main disk usage is 331 MB for the two-electron atomic integrals and 7 MB for the transformed integrals. The remainder of the disk usage is for holding temporary

information, loops and the CI vectors. The memory requirement does not decrease when more processors are used, but the amount of disk used per node does.

Table 1 shows the timing results for one iteration of the MCSCF energy on one to five nodes. The times reported are from the master node. Sequential steps are marked as such in the table. The one node timing shows that the transformation is the sequential bottleneck for this example.

Ten passes are needed to perform the transformation on 1 node. Thus, in this particular example, the number of passes could not be made evenly divisible by three or four and so ideal load balance is not achieved for 3 and 4 nodes. The transformation step has speedups (time on one processor/time on p processors) of 2.0, 2.5, 3.3, and 5.0 for 2, 3, 4, and 5 nodes, respectively. The corresponding efficiencies ($100 * \text{speedup}/p$) are 100%, 83%, 83%, and 100%. Of course, the low efficiencies for 3 and 4 nodes are from the master having to perform extra passes that some of the other nodes do not have to do.

While most of the other steps in this example use a negligible amount of time, steps 9, 10, and 11 are worth some comment. The transformed two-electron integral contributions to the Lagrangian and orbital hessian show only small time decreases when more nodes are applied. Since each node must read in the density from disk and put it into memory, this is a sequential part of the step that cannot be avoided. However, there may be other contributions to the poor scalability of this step.

The one-electron integral contributions to the Lagrangian and orbital hessian actually increase in time from 1 to 3 nodes. Only the master node is calculating the

Table 1. Timing information from the master node in seconds for H_2NCH for 1 to 5 nodes

Step[a]	1	2	3	4	5
1. Guess[b]	3.8	4.1	5.3	5.2	5.1
2. AO int[b]	391.9	392.0	391.5	391.0	391.0
3. DRT[b]	0.5	0.5	0.6	0.6	0.6
4. Trans	1539.1	764.5	616.2	461.0	304.7
5. Sort	1.2	0.7	0.6	0.5	0.4
6. GUGA loops	0.1	0.1	0.1	0.1	0.1
7. Diag	0.1	0.1	0.1	0.1	0.1
8. DM2	0.1	0.1	0.2	0.2	0.2
9. Lag+hess					
2 e^-[c]	12.1	9.1	8.7	8.0	7.2
1 e^-[d]	4.5	11.5	18.1	18.3	18.1
10+11. NR[b]	25.2	27.0	25.4	25.4	25.4
iter.[e]	1582.4	813.1	669.4	513.5	356.1
eff.[f]		97%	79%	77%	89%

[a] The steps correspond to: 1) obtaining the initial guess, 2) calculating the atomic integrals, 3) generating the distinct row table, 4) transforming into the MO basis, 5) sorting the transformed integrals, 6) calculating GUGA loops 7) diagonalizing **H**, 8) calculating the electron density matrices, 9) forming the Lagrangian and the orbital hessian, 10) forming the augmented orbital hessian and 11) solving the NR equations.

[b] This is a sequential step.

[c] Transformed two-electron integral contribution.

[d] Transformed one-electron integral contribution plus global sum of Lagrangian and the orbital hessian.

[e] One iteration time – the sum of steps 4 through 11.

[f] The efficiency (speedup/number of processors) based on steps 4 through 11.

actual one-electron contributions. The extra time for 2 to 5 nodes is associated with the large global summations of the Lagrangian and orbital hessian whose timing is also included in this step. The orbital hessian, in particular, is quite large. Fortunately, the global summation time seems to level off after about 3 nodes. TCGMSG employs a clever algorithm to diminish the number of operations and communication required for a global sum. For details, the reader is referred to reference [10].

Steps 10 and 11 are sequential steps. Even though they consume only a small amount of CPU time in this example, Amdahl's Law shows that even small sequential bottlenecks quickly effect the efficiency.

Since the transformation is the major bottleneck for this example, the efficiency for one iteration (steps 4 through 11) mostly follows the efficiency of the transformation. In this example, very good overall efficiencies are seen even at 5 nodes, implying that even more nodes could effectively be used for this example. Of course, as is seen in the 5 node timing, as the transformation time becomes small the other steps will consume a larger percentage of the overall time. Therefore, the overall efficiency will decrease. At 5 nodes, the time for steps 9, 10, and 11 is about 14% of the total iteration time.

The second example is Ge_2F_4 using a 3-21G [19] basis set totaling 82 basis functions. The active space consists of four electrons in four orbitals which generates 20 CSFs (the same as the first example). This time, however, there are 48 core orbitals; therefore, the NR step as well as the integral transformation is the bottleneck for the calculation. This example requires 3.8 MW of memory and approximately 65 MB of disk on one node. The disk usage is 7 MB for the two-electron atomic integrals, 33 MB for the transformed integrals, 12 MB for the second order density, 8 MB for the sorted integrals, and 0.065 MB for the loops. Again, the memory requirement will be the same for each node, the AO integral list is duplicated on each node, and the transformed integral and loop disk space is evenly distributed across all of the nodes.

Table 2 shows the timing results for one iteration of the MCSCF energy on one to five nodes. The one node timing shows that indeed the transformation and NR solution are the bottlenecks for this example. The transformation step (4) has speedups of 2.3, 3.7, 6.2, and 7.6 for 2, 3, 4, and 5 nodes, respectively. While this seems strange (the speedups are actually larger than the number of nodes), the timing results given in the table are for the master only. The slave nodes are taking more work than the master; this imbalance results in speedups that appear to be greater than the theoretical 100% efficiency. One way to try to rectify this imbalance is to make the number of passes larger, but this would also require the atomic integrals to be read in or calculated (if using the direct option) more times than necessary. Thus, this is not a very attractive option. Since the load imbalance is not large, we have decided to implement the algorithm as described above.

It is interesting that several steps (the sorting, density generation, and the transformed one-electron integrals – steps 5, 8, and 9) actually increase in time from 1 to 3 nodes, instead of decreasing as would be expected. As noted in the first example, the time required to perform the large global sums is actually a significant percentage of each step. The large number of core orbitals gives relatively many occupied MO integrals needed in the sort (step 5) and many density elements in the density matrix generation (step 8) and increases the size of the orbital hessian (step 9). For example, the global sum operates on 614, 916 double precision elements in step 5. To send such a long data set across the network and sum uses a non-trivial amount of CPU time. Of course, the more nodes involved, the larger the number of additions. As is seen in the first two examples and will be seen in the third example,

Table 2. Timing information from the master node in seconds for Ge_2F_4 for 1 to 5 nodes

Step[a]	1	2	3	4	5
1. Guess[b]	6.4	6.7	7.1	7.2	7.1
2. AO int[b]	21.7	21.7	21.9	21.8	21.8
3. DRT[b]	1.4	1.4	1.5	1.5	1.5
4. Trans	237.9	104.4	63.6	38.4	31.4
5. Sort	9.3	14.9	22.6	23.0	21.1
6. GUGA loops	1.0	0.7	0.8	0.7	0.8
7. Diag	0.1	0.1	0.1	0.1	0.1
8. DM2	30.1	39.3	48.1	48.1	48.2
9. Lag+hess					
2 e^-[c]	51.4	37.0	30.2	26.9	23.9
1 e^-[d]	10.4	24.7	27.4	27.7	27.6
10+11. NR[b]	100.0	100.4	100.2	100.1	100.3
iter.[e]	441.2	321.3	292.8	264.0	253.4
eff.[f]		67%	50%	42%	35%

[a-f] See Table 1 for notes.

the global summation of the orbital hessian (step 9) is always large, but the other two steps (5, 8) are smaller when small numbers of core orbitals are used.

A large bottleneck in this example is the solution of the NR equations. When more than two nodes are used in this particular example, the NR step is the most time consuming portion of the calculation. This is obviously a part of the code that will need to be improved in the future.

The overall efficiencies for one iteration are reasonable only up to about 3 nodes. The time consuming, sequential NR step is the reason for the poor scalability beyond this point.

The third and final example is bicyclobutane, C_4H_6, using the 6-31G(d) basis set [20] giving 72 basis functions. There are 10 core orbitals and the active space is 10 electrons in 10 orbitals which generates 19, 404 CSFs. 1.3 MW of memory and approximately 380 MB of disk are required when run on 1 node. The major disk usage is 39 MB for atomic integrals, 13 MB for transformed integrals and 294 MB for the loops (10 MB for diagonal loops and 284 for off-diagonal loops). On five nodes, each node would require 39 MB for atomic integrals, 2.6 MB for transformed integrals and 58.8 MB for the loops (2 MB for diagonal loops and 56.8 MB for off-diagonal loops).

Timing information for this example is given in Table III. In this example, the bottleneck is clearly the CI diagonalization step and it remains the bottleneck even up to five nodes. The speedups for this step are 1.9, 2.7, 3.4 and 4.2 for 2, 3, 4, and 5 nodes, respectively, giving an 84% efficiency at 5 nodes. As noted in the algorithm section, these speedups are from the parallel formation of the matrix product ***HC*** and NOT from the small diagonalization during each iteration. The CI step requires 34 iterations to converge.

Steps 6 and 8 suffer from the problem of having only minimal parallelization in the CPU time. As noted above, the choice of how to parallelize loop generation was made to ensure that step 7, rather than steps 6 and 8 showed the greatest parallel efficiency. This is seen in the modest decrease in time when more nodes are applied to the problem. Certainly, the formation of loops will require attention in the future. Notice that since the number of elements involved in these steps is smaller than those of the first example (24, 645 sorted integrals in the current example as

Table 3. Timing information from the master node in seconds for bicyclobutane for 1 to 5 nodes

Step[a]	1	2	3	4	5
1. Guess[b]	0.8	1.0	1.3	1.3	1.3
2. AO int[b]	69.4	69.7	70.1	69.0	69.0
3. DRT[b]	1.0	0.9	1.0	1.0	1.1
4. Trans	118.0	57.9	48.8	27.6	22.8
5. Sort	2.0	1.2	1.2	1.1	0.9
6. GUGA loops	336.3	300.9	276.2	264.7	256.6
7. Diag	1904.0	1003.9	705.4	558.1	455.8
8. DM2	303.5	270.1	256.5	249.7	241.7
9. Lag+hess					
2 e$^-$[c]	74.3	59.0	57.4	50.0	49.3
1 e$^-$[d]	4.1	10.1	16.4	16.5	16.5
10+11. NR[b]	83.1	83.2	83.2	83.1	83.1
iter.[e]	2825.3	1786.3	1445.1	1250.8	1126.7
eff.[f]		79%	65%	56%	50%

[a–f] See Table 1 for notes.

compared to 614, 916 in the previous example) and since the total time is larger in this example, the global summations do not dominate the timing results. The parallel advantage, of course, is that the GUGA loop disk files are evenly distributed across all of the nodes reducing the storage demand on each node.

The transformed two-electron contributions to the Lagrangian and orbital hessian (step 9) still show poor speedup and the one-electron contributions still increase with the number of nodes. The orbital hessian is still very large for this example, and the global sum is certainly the cause for the increase in time.

The transformation step shows very good efficiencies in this example as it has in the other two examples. Even though it does not represent a large part of the calculation, parallelization of each step is important to achieve good overall speedups. In the same vein, since steps 10 and 11 are sequential they decrease the overall speedups even though they are not a large portion of the calculation.

The overall efficiencies for this example are quite good even up to 5 nodes. Obviously, to get better speedups larger test cases can be used. However, the real key to future improvement is to make the portions that are only minimally parallel (steps 6, 8, 10, and 11) more efficient.

4 Conclusions

Techniques for the parallelization of the integral transformation and the subsequent steps of the GUGA MCSCF calculation have been presented. The transformation and diagonalization steps show very good speedups. Other parts of the calculation (sorting of the transformed integrals, calculating the second order density, and forming the Lagrangian and orbital hessian) show only minimal speedups. Solution of the Newton–Raphson equation for the orbital improvement step has been identified as a sequential bottleneck.

Future improvements will involve obtaining more parallel content from the loop generation steps and to parallelize the Newton–Raphson step. Each of these will improve the overall scalability of the MCSCF iterations. Parallel computation of the analytic gradient [2] permits MCSCF geometry optimization, numerical

hessians, etc. proceed with the same overall efficiency as the generation of the MCSCF wavefunction itself.

Acknowledgements. This work was supported by a grant from the Air Force Office of Scientific Research (93-1-0105), a Department of Education GAANN fellowship to TLW and through the assistance of the Advanced Research Projects Agency. The IBM RS6000 workstations were purchased by Iowa State University.

References

1. Dupuis M, Watts JD (1987) Theor Chim Acta 71:91; Guest MF, Harrison RJ, van Lenthe JH, van Corler LCH (1987) Theor Chim Acta 71:117; Ernenwein R, Rohmer MM, Benard M (1990) Comput Phys Comm 58:305; Rohmer MM, Demuynck J, Bénard M, Wiest R, Bachmann C, Henriet C, Ernenwein R (1990) Comput Phys Comm 60:127; Harrison RJ, Kendall RA (1991) Theor Chim Acta 79:337; Cooper MD, Hillier IH (1991) J Comput-Aided Mol Des 5:171; Kindermann S, Michel E, Otto P (1992) J Comp Chem 13:414; Lüthi HP, Mertz JE, Feyereisen MW, Almlöf JE (1992) J Comp Chem 13:160; Brode S, Horn H, Ehrig M, Moldrup D, Rice JE, Ahlrichs R (1993) J Comp Chem 14:1142; also see (1993) Theor Chim Acta 255ff (No. 4–5)
2. Schmidt MW, Baldridge KK, Boatz JA, Elbert ST, Gordon MS, Jensen JH, Koseki S, Matsunaga N, Ngyen KA, Su S, Windus TL, Dupuis M, Montgomery JA Jr (1993) J Comp Chem 14:1347
3. Watts JD, Dupuis M (1988) J Comp Chem 9:158
4. Rendell AP, Lee TJ, Lindh R (1992) Chem Phys Lett 194:84
5. Harrison RJ (1991) J Chem Phys 94:5021; Harrison RJ, Stahlberg E (1992) CSCC Update 13:5; Schuller M, Kovar T, Lischka H, Shepard R, Harrison RJ (1993) Theor Chim Acta 84:489
6. Whiteside RA, Binkley JS, Colvin ME, Schaefer III HF (1987) J Chem Phys 86:2185; Covick LA, Sando KM (1990) J Comp Chem 11:1151; Wiest R, Demuynck J, Bénard M, Rohmer MM, Ernenwein R (1991) Comput Phys Comm 62:107
7. Ruedenberg K, Schmidt MW, Gilbert MM, Elbert ST (1982) Chem Phys 71:41; Ruedenberg K, Schmidt MW, Gilbert MM (1982) Chem Phys 71:51; Ruedenberg K, Schmidt MW, Gilbert MM, Elbert ST (1982) Chem Phys 71:65
8. Siegbahn PEM, Almlöf J, Heiberg A, Roos BO (1981) J Chem Phys 74:2384
9. Roos BO (1983) In: Diercksen GHF, Wilson S (eds) Methods in computational molecular physics. Reidel, Dordrecht, Holland, pp 161–187
10. Harrison RJ, Argonne National Laboratory, version 4.0.2, available by anonymous ftp in directory/pub/tcgmsg from host ftp.tcg.anl.gov. Harrison, RJ (1991) Int J Quantum Chem 40:847–863; Harrison, RJ (1993) Theor Chim Acta 84:363–375
11. Brooks B, Schaefer HF (1979) J Chem Phys 70:5092; Brooks B, Laidig W, Saxe P, Handy N, Schaefer HF (1980) Phys Scripta 21:312
12. Dupuis M, Chin S, Marquez A (1992) CHEM-station and HONDO. In: Malli GL (Ed) Relativistic and electron correlation effects in molecules and clusters. NATO ASI Series, Plenum Press, New York
13. Hollauer E, Dupuis M (1992) J Chem Phys 96:5220
14. Davidson ER (1975) J Comp Phys 17:87 Davidson ER (1983) In: Diercksen GHF, Wilson S (Eds) Methods in computational molecular physics. Reidel, Dordrecht, Holland, p 95
15. Brooks BR, Laidig WD, Saxe P, Goddard JD, Yamaguchi Y, Schaefer III HF (1980) J Chem Phys 72:4652; Brooks BR, Laidig WD, Saxe P, Schaefer III HF (1980) J Chem Phys 72:3837; Lengsfield III BH (1980) J Chem Phys 73:382
16. Siegbahn P, Heiberg A, Roos B, Levy B (1980) Phys Scripta 21:323
17. Yarkony DR (1981) Chem Phys Lett 77:634
18. TZV: Dunning TH (1971) J Chem Phys 55:715; diffuse: Clark T, Chandrasekhar J, Spiznagel GW, Schleyer PvR (1983) J Comp Chem 4:294; polarization: H exponents: p 2.0, 0.5; d 1.0 C exponents: d 1.44, 0.36; f 1.0 N exponents: d 1.96, 0.49; f 1.0
19. Binkley JS, Pople JA, Hehre WJ (1980) J Am Chem Soc 102–939; Dobbs KD, Hehre WJ (1986) J Comp Chem 7:359
20. Ditchfield R, Hehre WJ, Pople JA (1971) J Chem Phys 54:724; Hehre WJ, Ditchfield R, Pople JA (1972) J Chem Phys 56:2257