Theor Chim Acta (1989) 76: 187–193

Theoretica Chimica Acta

An efficient two-electron integral transformation for vector-concurrent computer architectures

Charles W. Bauschlicher, Jr.
NASA Ames Research Center, Moffett Field, CA 94035, USA

(Received March 18/Accepted March 23, 1989)

Summary. A matrix-multiplication based implementation of the two-electron integral transformation is compared to the "optimized" algorithm recently presented by Hurley, Huestis and Goddard. In spite of its poorer operation count, the matrix-multiplication based program runs significantly faster on the Alliant FX/8 than the code described by Hurley et al. Our code also uses much less memory, but requires more disk storage. Trade-offs between the requirements for disk storage, central memory, number of computing elements and CPU time are also discussed.

Key words: Two-electron transformation — Vectorization — Concurrency or multiprocessing

1. Introduction

Hurley, Huestis and Goddard [1] (HHG) recently described their implementation of a two-electron-integral transformation procedure for the Alliant FX/8 computer. In order to decide on the best procedure, HHG first analyzed various schemes for their approximate floating point operation count. They adopted what they called the "modified full index symmetry method" since it has the smallest operation count. They further noted that in this method the inner loop was executed concurrently on all the processors. As the inner loop is also vectorized, they were able to make use of both the vector and concurrent features of the Alliant.

Analyzing the various implementations based solely on operation count usually leads to the most efficient code on a scalar machine. However, on vector computers all operations are not equivalent, and therefore the analysis of operation count and the level of vectorization are coupled. We have developed

two transformations, the first for the CRAY computers based on matrix multiplication, and a second for the CYBER 205 based on the linked triad operation $Y = Y + a \times X$ (commonly called a SAXPY or DAXPY [2]). The CYBER 205 implementation is similar to that of HHG and holds the transformed integrals in memory. The implementation for the CRAY is based on the Yoshimine method [3] and does not require significant core storage. In this article we report on the performance of our transformation on the Alliant. The matrix multiplication based transformation was dismissed by HHG due to its operation count, but we show that it yields superior performance to that reported by HHG.

The code designed by HHG minimizes disk storage, but requires a very large central memory as all the transformed integral are held in core. On a machine with several CPUs (each main CPU on the Alliant is called a computing element, or CE), it is logical to allocate the memory and disk resources based on the number of CEs actually being used by each job. For example, a job that used only one of n CPUs should be allowed to use only $1/n$ of the memory and $1/n$ of the disk space. Since the results presented by HHG showed that their transformation became rather inefficient when the number of CEs was increased from 4 to 8, it would probably be best to limit it to run on 4 CEs and hence use only half the memory. Thus when a large calculation is to be performed using the HHG algorithm, one must choose between two unpleasant options: either using all of the CEs inefficiently for the transformation, or using only 4CEs for the transformation, and therefore forcing the remaining CEs to run a limited subset of jobs that require little memory. If a sufficient number of jobs requiring little memory do not exist, the latter option will also result in inefficient use of some CEs. It is well known that it is possible to design a transformation that requires much less memory, but more disk storage than that described by HHG. Since either memory or disk usage might become the limiting factor, it is clear that CPU time might not become the sole factor in designing a program. In this manuscript we discuss such trade-offs as well as the design of the matrix multiplcation based transformation.

2. Two-electron-integral transformation

The first step in *ab initio* electronic structure calculations is to compute the two-electron integrals over atomic (or symmetry) orbitals (AOs), where the two-electron integrals are given as

$$A_{ijkl} = \int \chi_i(1)\chi_j(1)\frac{1}{r_{12}}\chi_k(2)\chi_l(2)\,d\tau.$$

For a basis set of size n, it is clear from this formula that fewer than n^4 integrals are unique. The list of the $n^4/8$ unique integrals can be formed by ordering the integrals such that $i \geqslant j$ and $k \geqslant l$, and the compound index ij is greater than or equal to the compound index kl, where a compound index is defined as $ij = i * (i-1)/2 + j$. In principle it is possible to directly solve the correlation problem using these integrals; however, in practice it leads to an intractable

problem. Therefore in a preliminary calculation, an optimal set of molecular orbitals (MOs), ϕ, are determined, where

$$\phi_\alpha = \sum_{i=1}^{n} \chi_i C_{i\alpha}.$$

Then an approximate solution to the correlation problem is obtained using these molecular orbitals. However, in most of these treatments it is beneficial to transform the two-electron integrals from the AO to MO basis set

$$Z_{\alpha\beta\gamma\delta} = \sum_{ijkl} C_{i\alpha} C_{j\beta} A_{ijkl} C_{k\gamma} C_{l\delta}.$$

The one-electron integrals, H, are transformed in a similar manner

$$Y_{\alpha\beta} = \sum_{ij} C_{i\alpha} H_{ij} C_{j\beta}.$$

Since the one-electron transformation converts the labels ij into $\alpha\beta$ it is also called a two-index transformation. If the one- and two-electron integrals are transformed one at a time, the processes are of order n^4 and n^8, respectively. However, it is clear that all of the one-electron integrals can be transformed at one time by two matrix multiplications

$$Y = C^T H C$$

thus reducing the process to n^3. In a similar manner, the two-electron transformation can be reduced to n^5. This is discussed in detail by HHG, and has been discussed previously by other authors [3–6], and we do not repeat it in detail here.

We use the methods described by Yoshimine [3] and Bender [4]. The Bender method is similar to that discussed by HHG and under some circumstances we use it when there is sufficient space to hold the transformed integrals in memory. In the Yoshimine algorithm the $(ij|kl)$ integrals are transformed to $(ij|\alpha\beta)$ in a series of two-index transformations. The integrals are then transposed to $(\alpha\beta|ij)$ and transformed to $(\alpha\beta|\gamma\delta)$. In this way the two-electron transformation is reduced to a series of one-electron-like transformations. Since the integrals are transformed in a series of steps, at no time are all the transformed integrals held in memory and therefore only little memory is required. While the memory required is rather modest, the half-transformed integrals, $(ij|\alpha\beta)$, must be held on disk. Thus the requirements for this transformation are very different from that of HHG. We now discuss the organization of our transformation.

3. Code organization

In our implementation, we store the two-electron integrals $(ij|kl)$ one row per record, with all kl values for each ij, that is we ignore the $(ij)-(kl)$ permutational symmetry. The row can be stored with zeros included or with only the non-zeros (and information about their position in the uncompressed row). In all of the

results reported in this article, the rows are stored with the zeros. The transformation code is then divided into two half transformations. The first half is organized as follows:

1) a kl row of two-electron integrals is read,

2) a square matrix of integrals is formed from the lower triangle of kl integrals,

3) the $(ij|kl)$ integrals are transformed to $(ij|\alpha\beta)$ with two matrix multiplications,

4) the square matrix of transformed integrals is compressed into a lower triangle by discarding the redundant integrals, and

5) the transformed integrals are moved into bins and when the bin is full it is written to disk. This is the first half of the transposition of the integrals to $(\alpha\beta|ij)$.

Steps 1–5 are repeated for each ij.

The second half of the transformation is organized as follows:

6) a batch of integrals are moved from disk into memory finishing the transposition of the half-transformed integrals,

7) for each $\alpha\beta$ row currently in memory, the transform to $(\alpha\beta|\gamma\delta)$ is completed using two matrix multiplications, and

8) The block of fully transformed integrals are written to disk.

Steps 6–8 are repeated until all the integrals are transformed.

Unlike HHG we use REAL*8 variables throughout. This is important when very accurate calculations are to be performed, since as the basis sets become more complete, higher accuracy in the transformed integrals is required to avoid numerical problems. The program is all in FORTRAN except for the matrix multiplication where we use the Alliant library [7] routine "matmul". Using a FORTRAN matrix multiplication has only a small (about 3%) effect on the time required. (Note that this is very different from the CRAY computers where the library matrix multiply is much faster than the FORTRAN code. Further, unrolling the matrix muliplication on the Alliant degrades performance; this is also different from the CRAY, where unrolling greatly improves the performance of the FORTRAN code.) We compile the program with optimization for vectorization, scalar and concurrency. We run it on one, two and four CEs (our system has only four CEs). The time (user plus system) is determined using the library routine [7] "etime". We report some breakdown of the timing using "gprof" [7].

Memory usage can always be traded for disk storage, but until recently this was not really practical as central memories were quite limited in size. Now, however, computers have very large memories and therefore codes can be designed to use this feature. In the matrix multiply based transformation, the half-transformed integrals instead of the transformed integrals could be held in

memory, but this would require more storage than the Bender or HHG methods, especially in those cases where the number of MOs is significantly less than the number of AOs. Our method, in addition to using disk space to store the half-transformed integrals, also uses twice the storage of the AO integrals due to the elimination of the $(ij)-(kl)$ symmetry. However, as we show below it uses only about half the CPU time. That is, our code was developed for a system with a very large disk capacity where CPU time is the limiting factor.

4. Results

Our code is most easily thought of as a series of two-index transformations. Each two-index transformation is two matrix multiplications which are of order n^3. All other steps are of order n^2. In addition to these two steps, there is system time and some time not accounted for with gprof. Therefore we break the timing down into the three steps, denoted n^2, n^3 and overhead. The timings for the transformation of the two-electron integrals for C_2F_4 and C_2F_6 are summarized in Table 1 along with the results of HHG. For both C_2F_4 and C_2F_6, the time for our implementation on one CE is less than half that reported by HHG. Of the total time in our transformation 78–82% is spent in matrix multiplication. When the number of processors is increased, the time in matrix multiplication decreases with only a small loss of efficiency. However, the overhead and n^2 steps do not show the same decrease. This is perhaps not too surprising since the IO, and hence the sort, is not easily done in parallel. The net result is that the overall transformation shows a significant overhead with concurrency. For two processors we have an efficiency of 87% and for four it has dropped to 70%. In the HHG algorithm the integrals are not sorted, but held in core, and this scheme therefore shows a performance improvement with number of processors more like that observed in the matrix multiplication step in our program. The efficiency is 97%, 88% and 68%, so even their algorithm shows a marked decline in efficiency when the number of CEs is expanded from 4 to 8. Clearly neither our code nor that of HHG would be suited to a machine with a very large number of processors.

There is one additional point that we wish to make. We have shown that by using a different algorithm the two-electron-integral transformation can be performed in about half the time reported by HHG. However, up to this point we have ignored the use of symmetry. On a scalar machine one might process symmetry zeros and accidental zeros in the same manner, but to achieve the best vectorization the testing of zeros must be eliminated. (Some improvement could be obtained by using a matrix multiplication that accounts for the sparseness of the integral or coefficient matrices, but it does not appear possible to account for sparseness in both matrices efficiently.) However, it is still possible to make use of the symmetry zeros by sorting the integrals by symmetry blocks, and ordering them within each block in a manner similar to the no symmetry case. The integrals are then transformed one symmetry block at a time; in this way one large transformation is reduced to a series of small transformations. For C_2F_4,

Table 1. The transformation time, in seconds, on the Alliant FX/8

	1 CE		2 CE			4 CE		
	Time	%[a]	Time	%	Ratio[b]	Time	%	Ratio
	C_2F_4 54 basis functions Present work							
n^3	205	78	104	69	1.96	54	57	3.79
n^2	43	16	31	20	1.38	25	26	1.73
Overhead	14	5	16	11	0.83	17	18	0.81
Total	261		151		1.72	95		2.74
HHG[c]	544		277		1.96	154		3.53
HHG/PW[d]	2.08		1.83			1.62		
	C_2F_6 72 basis functions Present work							
n^3	859	82	421	70	2.04	221	60	3.88
n^2	136	13	103	17	1.32	80	21	1.70
Overhead	55	5	77	13	0.71	71	19	0.77
Total	1050		601		1.75	372		2.82
HHG	2170		1132		1.92	625		3.47
HHG/PW	2.07		1.89			1.68		

[a] Percent of the total time
[b] Ratio is the time for 1 CE divided by the time on n CEs
[c] HHG [1]
[d] Ratio of total time for HHG to that in present work

if D_{2h} symmetry is used, the transformation time is reduced to 34 seconds on one CE. Note that when symmetry is used, this case is so small that only 4 seconds are spent in matrix multiplication. Thus in a very large case the use of symmetry will yield an even larger savings. This illustrates that while using an algorithm that vectorizes better can make a factor of two improvement, it is actually far better to design the program to make use of symmetry, in spite of some added complexity, since this can represent an order of magnitude improvement.

As noted above we have also implemented a modified version of the Bender method (including symmetry), and our times are similar to those reported by HHG. That is about a factor of two slower than the matrix multiplication based approach. It is interesting to note that on a CRAY X-MP/48 computer (with a 9.5 ns clock) the 54 basis set transformation requires only about 10 seconds using either the matrix multiplication or Bender method. This is about 4 times faster than the transformation designed for the CRAY-1 by Saunders and van Lenthe [6] (note we have scaled their CPU time by the ratio 9.5/12.5 to account for the difference in cycle time between the CRAY-1 and the CRAY X-MP). Their algorithm, like that of HHG, is based on using the $(ij)-(kl)$ permutational symmetry, but requires the same storage of half-transformed integrals as our code. The decision to implement either the matrix-multiplication based method or the Saunders and van Lenthe on a CRAY would depend on the problems to be run and system configuration. For example if one planned to use large AO

basis sets, but to transform to only limited MO sets on a system that had limited disk storage, the Saunders and van Lenthe method would probably be the best. Thus even on a supercomputer one might compromise CPU time for disk storage: the ideal implementation depends on the specific configuration of the computer to be used.

In performing calculations on a CYBER 205, we have found that our matrix multiplication based algorithm worked poorly even for modest sized matrices because of the large vector overhead. Therefore we have implemented a modified version of the Bender method, since this produces SAXPY operations with long vector lengths that are ideally suited for the CYBER 205. This method generally required significantly less time than the matrix multiplication based approach on the CYBER 205. Thus, the "optimal" transformation varies due to machine hardware as well as the specific configuration of memory and disk resources. Since our code has both the Bender and Yoshimine transformation, the optimal performance can be obtained on a variety of current computers. Furthermore, we make explicit use of symmetry, as the formation of two-electron integrals over symmetry orbitals increases the cost of integral generation only slightly [8], introduces almost no overhead into calculations, and can represent an order of magnitude savings in computer time. In spite of this flexibility, the transformation is one of the simplest codes in our program system.

5. Conclusion

We have shown that a different implementation for the two-electron integral transformation than that reported by HHG yields superior performance even though it has a higher operation count. This is an illustration of the need to consider the combination of the operation count and extent of vectorization when designing a code for modern vector computers. In addition to the question of CPU performance, disk storage and memory requirements also must be considered in designing a transformation. Thus the "optimal" transformation can vary greatly between different computers and even between different configurations of the same computer.

References

1. Hurley JN, Huestis DL, Goddard WA (1988) J Phys Chem 92:4880
2. Lawson C, Hanson R, Kincaid D, Krough F (1979) ACM Trans Math Software 5:308
3. Yoshimine M (1971) In: Lester WA (ed) Proceedings of the Conference on Potential Energy Surfaces in Chemistry. Report RA-18, IBM Research Laboratory, San Jose, CA, p 87
4. Bender CF (1971) J Compt Phys 9:547
5. Elbert ST (1978) In: Moler C, Shavitt I (eds) Numerical algorithms in chemistry: algebraic methods. LBL 8158, Lawrence Berkeley Laboratory, University of California, Berkeley, p 129
6. Saunders VR, van Lenthe JH (1983) Mol Phys 48: 923
7. FX/FORTRAN Language Manual, Alliant Computer Systems Corporation, Littleton, Mass (1987)
8. Almlöf J (1974) Molecule Program Description. University of Stockholm, Institute of Physics, Report 74-29