Theor Chim Acta (1996) 93: 317–331

Theoretica Chimica Acta

© Springer-Verlag 1996

Parallel direct four-index transformations

Adrian T. Wong[1], Robert J. Harrison[1], Alistair P. Rendell[2]

[1] Pacific Northwest Laboratory[1], P.O. Box 999, Richland WA, 99352, USA

[2] Daresbury Laboratory, Daresbury, Warrington WA4 4AD, UK

Received June 26, 1995/Final revision received November 10, 1995/Accepted November 10, 1995

Summary. Two parallel direct integral transformation algorithms are presented. Specific attention is directed to producing transformed integrals containing at least two "active orbital" indices. The number of active orbitals is typically much less than the total number of molecular orbitals reflecting the requirements of a wide range of correlated electronic structure methods. Sample direct second-order Møller–Plesset theory calculations are reported. For situations where multipassing of the integrals is required, superlinear speedup is obtained by exploiting the increase in global memory. As a consequence, for morphine in a 6-31G basis, a speedup of over 25 is observed in scaling from 32 to 512 processors.

Key words: Parallel – Four-index transformation – Møller–Plesset

1 Introduction

The transformation of two-electron integrals from the atomic orbital (AO) to molecular orbital (MO) representation is a fundamental requirement of dynamically correlated and muticonfiguration self-consistent field (MCSCF) methods. Originally this was performed by computing the AO integrals, writing them to disk, and then processing them in a series of one-index transformations and sorts – an input/output (I/O) intensive procedure. When disk space is limited and/or when the I/O rate is poor relative to processor speed, an alternative *direct* algorithm may be employed. In analogy to direct SCF, a direct transformation will recompute the AO integrals as required rather than retrieve them from disk. A number of different direct transformation strategies may be considered [1], some of which have been implemented in the framework of second-order perturbation theory (MP2) [2–6].

On massively parallel processors (MPPs) the ratio of compute to I/O performance has traditionally been very large and, on a given machine, tends to increase with the number of processors being used. Thus, it would appear more profitable to

[1] Pacific Northwest Laboratory is operated for the U.S. Department of Energy by Battelle Memorial Institute under contract DE-AC06-76RLO 1830

[2] *Current address*: Supercomputer Facility, Australian National University, Canberra, ACT 0200, Australia

investigate parallelization of a direct rather than conventional transformation. The earlier parallel four-index transformations [7–11] used either integrals stored on disk or assumed that all AO integrals could be held in aggregate memory. Clearly, for a practical code there is a need to address the issues of memory limitation and direct AO integral generation in the transformation. Several recent parallel algorithms reported [5, 6, 12] have used a direct method.

In this paper we discuss possible parallel direct integral transformation strategies and implement two viable algorithms. Specific attention is given to the special case where only a subset of the transformed two-electron integrals containing at least two "active orbital" indices are produced. The range of the active orbital indices is assumed to be much smaller than the range of the full MO indices, although it could obviously be expanded to cover all MO indices and thereby produce a full list of transformed integrals. The rationale for working within this restriction derives from the fact that many correlated methods either do not require other classes of transformed integrals, e.g., MP2 or the complete active space self-consistent-field method (CASSCF), or contributions from other classes of integrals can be evaluated directly from AO integrals, e.g., the single and double-excitation coupled-cluster method (CCSD) [13, 14].

In the following section we briefly review direct integral transformations noting potential problems in migrating these algorithms to MPPs. More detailed discussion concerning limited integral transformations has been given in several earlier articles [1, 15, 16]. In Sect. 3 we consider in detail aspects associated with parallelization, highlighting potential communication bottlenecks and developing a rudimentary performance model for our two parallel algorithms. The MP2 method, which is a special case of a limited direct integral transformation, is covered in Sect. 4. Section 5 presents observed parallel performance for two different test molecules.

2 Direct integral transformations

In this work, only the subset of integrals which contain at least two active orbital indices will be considered. The active orbitals have the usual definition for MCSCF but are defined as the correlated occupied orbitals for MP2 and other dynamically correlated methods. This subset of integrals are grouped into Coulomb (J^{ij}) and exchange (K^{ij}) operator matrices,

$$J^{ij}_{pq} = [pq|ij], \tag{1}$$

$$K^{ij}_{pq} = [pi|qj], \tag{2}$$

where the integrals are written in Mulliken notation, $i, j, \ldots$ denote active orbital indices and $p, q, \ldots$ arbitrary MO indices. Using N as the number of basis functions and n as the number of active orbitals, typically $n \ll N$. For the Coulomb integrals, transformation from the AO to MO basis requires the evaluation of the following fourfold summation:

$$[pq|ij] = \sum_{\mu\nu\lambda\sigma} C^p_\mu C^q_\nu C^i_\lambda C^j_\sigma [\mu\nu|\lambda\sigma], \tag{3}$$

where the Greek indices, $\mu, \nu \ldots$ denote AOs and C is the matrix of MO coefficients. The expression for the exchange integrals, $[pi|qj]$, is identical except for

interchanges of indices. The transformation is most efficienctly performed as four consecutive quarter transformations [17]:

$$[\mu\nu|\lambda j] = \sum_{\sigma} C_{\sigma}^{j}[\mu\nu|\lambda\sigma], \tag{4}$$

$$[\mu\nu|ij] = \sum_{\lambda} C_{\lambda}^{i}[\mu\nu|\lambda j], \tag{5}$$

$$[\mu q|ij] = \sum_{\nu} C_{\nu}^{q}[\mu\nu|ij], \tag{6}$$

$$[pq|ij] = \sum_{\mu} C_{\mu}^{p}[\mu q|ij], \tag{7}$$

since the total computational cost will then scale as the fifth power of the number of basis functions (N^5) rather than N^8 which might be implied from Eq. (3). An optimal integral transformation will seek to minimize the prefactor in front of N^5, while maintaining reasonable memory and disk requirements. In most implementations, the transformation is divided into two halves; performing transformations (4) and (5) initially and then transformations (6) and (7) separated by an intervening *supermatrix transposition*, i.e., a sort of the integrals from all i, j for fixed $(\mu\nu)$ to all μ, ν for fixed (ij). Since the latter half transformation is trivially effected after the transposition, most of our attention will be directed toward optimizing the first half transformation and transposition. As suggested by the order of Eq. (4)–(7), for the limited transformation to J and K it is advantageous to transform to the active MO indices in the first and second quarter transformations. Since $n \ll N$, this ordering significantly reduces the size of the intermediate quantities and therefore the cost of subsequent transformation steps.

In a conventional disk-based transformation, there is a considerable storage advantage to be gained by using a transformation algorithm which works from a minimal list of AO integrals. In other words, both the *overlap* ($[\mu\nu|\lambda\sigma] = [\mu\nu|\sigma\lambda]$) and *supermatrix* ($[\mu\nu|\lambda\sigma] = [\lambda\sigma|\mu\nu]$) symmetry of an integral is exploited. Obviously, this is also an advantage in a direct scheme as it eliminates redundant computation of integrals. However, since the evaluation of the AO integrals scales as N^4, while the transformation scales as N^5, in the limit of a very large basis the computation of redundant integrals may not be significant. Furthermore, exploiting the full permutational symmetry may inhibit parallelism and an algorithm with some redundant AO integral computation may scale better in practice.

A scheme which does compute J and K given a minimal list of AO integrals has been outlined by Werner and Meyer [15]. This algorithm is particularly efficient on a single processor, but relies on the availability of sufficient memory to store J, K and a temporary array of dimension nN^2. For J and K this assumption is generally justified as it is often assumed in subsequent correlated calculations; the main exception is the direct MP2 algorithm where some paging of K is usually implemented. The temporary array, however, is more problematic; its size means that on most MPPs it cannot be replicated in local memory on each processor but must be distributed. Repeated zeroing and subsequent accumulation of data into this array implies some degree of synchronization between the processors using it and, consequently, a potential load balancing problem. In view of this difficulty and the inability to implement paging into the computation of K this algorithm is probably not ideal for an MPP.

```
       create and zero J and K global arrays
P >    synchronize
       do μ = 1, n_sh
          do ν = 1, μ
P >          if ((μν) is my task) then
                do λ = 1, n_sh
                   do σ = 1, λ
                      compute AO integral block [μν|λσ]
                      [μν|λi] = [μν|λi] + Σ_σ C^i_σ [μν|λσ]
                      [μν|σi] = [μν|σi] + Σ_λ C^i_λ [μν|λσ]
                   end
                end
                [μν|ji] = Σ_ζ C^j_ζ [μν|ζi]
                scatter [μν|ji] into global J array
                do i = 1, n
                   do j = 1, i
                      [μj|ζi] = Σ_ν C^i_ν [μν|ζi]
                      [νj|ζi] = Σ_μ C^i_μ [μν|ζi]
                      accumulate [μj|ζi] and [νj|ζi] into global K array
                   end
                end
P >             get my next task
P >       end
        end
       end
P >    synchronize
       do i = 1, n
          do j = 1, i
P >          if (J^ij in local memory)
                J^ij_pq = Σ_μν C^p_μ C^q_ν J^ij_μν
P >          end
P >          if (K^ij in local memory)
                K^ij_pq = Σ_μν C^p_μ C^q_ν K^ij_μν
P >          end
          end
       end
P >    synchronize
```

Parallel constructs indicated by P >

Fig. 1. Twofold redundant parallel direct transformation algorithm for J and K

An alternative algorithm with smaller memory requirements for temporary arrays, and which is also amenable to paging of J and K can be produced by not exploiting the supermatrix symmetry of the integrals, i.e., the integrals are computed twice. This scheme, as implemented in a direct context, is given in Fig. 1.

Essentially, it is a generalization of the direct MP2 algorithm of Head-Gordon et al. [3, 4] to compute J in addition to K with a redefinition of the virtual orbital range to include all MO indices. For optimal AO integral generation, the outer four loops are over shell indices, where a shell consists of all angular components for a given radial function and there are a total of n_{sh} shells. Using S^{μ} to denote the number of components for shell μ, the integrals are generated in blocks of dimension $s^{\mu}s^{\nu}s^{\lambda}s^{\sigma}$. Partial contributions to the first-quarter integral transformation are evaluated as the integral shell blocks are computed, accumulating data into a temporary array of size S^2nN, where S is the maximum shell size. For shells with a small number of angular components the performance of this *progressive* transformation may be relatively poor and, subject to available memory, several integral shell blocks should be grouped together before effecting a partial contribution to the first index transformation.

Completion of the loops over λ and σ (Fig. 1) yield an intermediate array of integrals, $[s^{\mu}s^{\nu}|\zeta i]$ where the dummy index ζ spans all AO indices. At this point, the Coulomb half-transformed integrals, $[s^{\mu}s^{\nu}|ji]$, can readily be generated with a single-matrix multiplication. However, only a partial contribution to the half-transformed exchange integrals, $[s^{\mu}j|\zeta i]$ and $[s^{\nu}j|\zeta i]$, can be effected. These integral quantities are not complete until the end of the μ and ν loops. Note that the ζ index can be transformed immediately to MO indices although there is no computational gain unless the number of MOs is substantially less than the size of the AO basis (e.g., MP2).

The third and fourth indices of J and K are transformed using a pair of matrix multiplications positioned after all loops defining the integral generation are complete. As mentioned earlier, this requires J and K to be arranged in memory such that for a given MO pair index (ij) all μ, ν are available, rather than the fixed $(\mu\nu)$ all i, j ordering produced after the second index transformation. Either an explicit supermatrix transposition must be performed prior to the final transformations or an implicit transposition performed by scattering the blocks of data as they are produced after the second index transformation into the correct final ordering. For J either strategy is suitable, however, for K an implicit transposition implies that inside the loop over μ and ν a contribution to K of size n^2N must be scattered (and accumulated) in memory. For this implicit transposition the data traffic scales as $n_{sh}n^2N^2$, whereas for an explicit transposition it scales as n^2N^2. In terms of data movement the explicit transposition is preferred, however, the implicit transposition has the advantage of smaller memory requirements since the half-transformed integral blocks are accumulated directly into their final destination.

3 Parallel direct transformations

For simplification we will initially assume that the number of Coulomb and exchange integrals, $O(n^2N^2)$, is sufficiently small such that they can be held within the global memory of the parallel machine. Given that there are currently MPPs with total memory capacities in excess of 20 GB, this will be true for even quite large systems. However, in the next section and in reference to MP2, we show how paging can trivially be incorporated into our parallel algorithm. It is important, at this stage, to distinguish between data stored in *global* memory from that stored in *local* memory; on a distributed memory, architecture access to global data is considerably slower owing to the interprocessor communication involved. For virtual shared-memory architectures, e.g., Cray T3D and KSR, where access to

global data may be implemented by the operating system or hardware, a similar difference in memory latencies also applied. Furthermore, while the size of the global memory grows with the number of processors, the local memory capacity is fixed and, hence, represents a more stringent constraint since it is independent of the number of processors employed. Therefore, the primary design goals, as far as the memory issues are concerned, are to minimize the number of global memory transactions and to use the smallest amount of local memory possible. The J and K operators are distributed in global memory in a matrix structure where a processor holds entire J^{ij} or K^{ij} matrices for one or more index pairs (ij). Although this distribution is advantageous given the generation and use of J and K, it does imply a minimum of $O(N^2)$ memory per processor. This local memory assumption will underly our parallelization scheme, and enables us to replicate the MO coefficient matrix on each processor.

Assuming that J and K are stored in a global shared-memory region, parallelization of the direct four-index transformation given above is straightforward as shown by Fig. 1. The compound shell index $(\mu\nu)$ designates a parallel task which involves the generation of half-transformed integrals for fixed $(\mu\nu)$, i.e., all computations inside the outer two loops. This yields $O(n_{\mathrm{sh}}^2)$ parallelism which is likely to be sufficient for most platforms of interest. In a *static* load distribution, the list of $(\mu\nu)$ tasks would be evenly divided between the processors from the outset. However, the time spent on different tasks varies widely depending on integral sparsity and nature of the basis functions that constitute shells μ and ν. Therefore, these tasks are allocated *dynamically* using a shared counter which maintains optimal load balancing. As noted earlier, starting the third index transformation requires completion of the summation inherent in the second integral transformation, and thus is necessary to synchronize the processors outside of the $(\mu\nu)$ loops. To minimize load-balancing problems associated with this synchronization, and in analogy to parallel direct SCF codes, it is best to process the task list in descending order of expected processing time. For example, the shells may be reordered in descending shell length since the tasks with high angular momentum shells are more computationally demanding. Finally, the third and fourth index transformations are parallelized according to locality of the distributed J and K. The above parallel algorithm is attractive since it is fully load balanced and only requires three synchronizations: one to ensure that J and K are initialized to zero, one after all AO integral generation is complete and before the third index transformation, and one after the fourth index transformation. Only the second synchronization has any substantial effect on the parallel efficiency since it is MIMD at this point and processors may have to wait for other processors to finish their allotted tasks. The first and third synchronizations have little or no effect since all processors are already nearly synchronized at these stages.

The principal local memory requirements for the above algorithm are

1. An array of size S^4, which is used to hold the AO integrals as they are generated. For a basis containing Cartesian f functions this will be of size 10^4.
2. A temporary array of size $S^2 nN$, used to hold the quarter-transformed integrals, $[s^\mu s^\nu|\zeta i]$.
3. The MO coefficient matrix, and the equivalent of 1–2 other matrices of the same size (N^2) which are used as temporary storage.

The last two requirements are likely to limit the system size which can be treated on a given machine. Taking 64MB of available memory per processor as an example, we anticipate being able to perform calculations in the region of 800 basis functions.

The parallelization scheme outlined above does, however, rely on the availability of a large global shared-memory region to store J and K. Recognizing the general utility of such a concept, Nieplocha et al. have devised a set of Global Array (GA) tools [18] that facilitate the manipulation of global data objects on distributed and virtual shared-memory architectures. These tools incorporate the simplicity of the shared-memory model, obviating the need for explicit message passing, while maintaining the distinction between global and local memory. This concept of a hierarchical memory structure, or non-uniform memory access (NUMA), already exists on many MPPs, but the GA tools provide a much greater degree of portability. The utility of the GA tools has already been demonstrated in a variety of computational chemistry applications on a number of platforms [19].

Using the GA tools to store J and K and the parallelization scheme outlined above, the critical issue becomes the movement of local data into the formally distributed J and K data objects. This occurs when the half-transformed integrals are placed into the final arrays pending the third index transformation. Since there is a need to move data to remote processors, this data movement can be combined with the supermatrix transposition required before the third and fourth index transformation. This implicit transposition is compatible with dynamic load balancing and has the advantage of overlapping the communication with the associated AO integral generation and transformation. The data movement for J scales as $n^2 N^2$. However, for K the parallelization scheme implies that different processors will sum different contributions into the same element of the half-transformed K matrix. In this case the communication scales as $n_{\text{sh}} n^2 N^2$, which is identical to the serial implicit transposition of K discussed earlier.

It is worthwhile noting that the above algorithm and the V1 algorithm of Nielsen and Seidl [6] derive from identical sequential algorithms. However, the resultant parallel algorithms are quite different, especially in the definition of parallel tasks. The differences perhaps most effectively demonstrate the advantages of using a shared-memory programming model. The earlier algorithm requires task synchronization and global communications with a fine task granularity while in this work the algorithm has a much higher degree of process asynchrony and a low volume of point-to-point communications with a coarser task granularity.

The overall scaling of the above algorithm will depend on the communication capability of the machine being used and the details of the calculation undertaken. The loop ordering $\mu\nu\lambda\sigma$ (varying slowest to fastest) is optimal for J, but problematic for K as it results in $O(N^5)$ data movement. The alternative is also to generate the integrals in the optimal order for the exchange transformation. This yields a sixfold redundancy in the AO integrals, with $[\mu\lambda|\nu\sigma]$ and $[\mu\sigma|\nu\lambda]$ computed together with $[\mu\nu|\lambda\sigma]$. In this case, the data movement for K scales as $O(N^4)$ since it avoids partial summations in the second index transformation. This algorithm, which is shown in Fig. 2, is labeled as *sixfold* while the earlier algorithm in Fig. 1 will be referred to as *twofold*. In Fig. 2, the generation of J and K has been coalesced into a single structure for brevity; however, it is actually implemented as two distinct loop structures since the J and K intermediates are independent. This has the advantage of reusing local memory, doubling the available parallelism, and reducing task sizes. The sixfold algorithm is similar to that proposed by Saebø and Almlöf [2], although since they only compute K they actually have a fourfold redundancy in the integral generation. In addition to the extra integral computational work, a further disadvantage of the sixfold algorithm is that Schwarz screening for K cannot exploit $(\mu\nu)$ sparsity.

```
        create and zero J and K global arrays
P >     synchronize
        do μ = 1, n_sh
          do ν = 1, μ
P >         if ((μν) is my task) then
              do λ = 1, n_sh
                do σ = 1, λ
                  compute AO integral block [μν|λσ]
                  [μν|λi] = [μν|λi] + Σ_σ C^i_σ [μν|λσ]
                  [μν|σi] = [μν|σi] + Σ_λ C^i_λ [μν|λσ]
                  compute AO integral block [μλ|νσ] and [μσ|νλ]
                  [μλ|νi] = [μλ|νi] + Σ_σ C^i_σ [μλ|νσ]
                  [μσ|νi] = [μσ|νi] + Σ_λ C^i_λ [μσ|νλ]
                end
              end
              [μν|ji] = Σ_ζ C^j_ζ [μν|ζi]
              scatter [μν|ji] into global J array
              [μj|νi] = Σ_ζ C^j_ζ [μζ|νi]
              scatter [μj|νi] into global K array
P >           get my next task
P >         end
          end
        end
P >     synchronize
        do i = 1, n
          do j = 1, i
P >         if (J^ij in local memory)
              J^ij_pq = Σ_μν C^p_μ C^q_ν J^ij_μν
P >         end
P >         if (K^ij in local memory)
              K^ij_pq = Σ_μν C^p_μ C^q_ν K^ij_μν
P >         end
          end
        end
P >     synchronize
```

Parallel constructs indicated by P >

Fig. 2. Sixfold redundant parallel direct transformation algorithm for J and K

A reasonable understanding of the comparative performance of our two algorithms can be obtained from a rudimentary model. For clarity, we consider only the generation of K, since it is the more problematic component, and ignore integral sparsity. Let α denote the time per floating point operation, β the average time per integral generation, and $\bar{s} \approx N/n_{sh}$ the average shell length. For the twofold

algorithm, the computation time for each task is given by

$$T^{2x}_{\text{compute}} = \bar{s}^2\left[\frac{1}{2}N^2\beta + 2N^2n\alpha + 2Nn^2\alpha\right]. \quad (8)$$

The first term is the cost of generating a block of $\bar{s}^2N^2$ AO integrals using overlap symmetry and the second and third terms are matrix multiplication cost of the first and second-quarter transformations. Multiplying Eq. (8) by the number of tasks, $\approx N^2/2\bar{s}^2$, recovers the total computation time which scales as $O(N^4\beta + N^4n\alpha + N^3n^2\alpha)$. In each task, $2\bar{s}Nn^2$ data is communicated with an aggregate volume of $O(n^2N^2n_{sh})$. The transposition is divided into messages of length $\bar{s}N$ for each (ij) and, using t_0 and t_1 to denote the latency and bandwidth, respectively, to a first approximation the communication time per task is

$$T^{2x}_{\text{comm}} = n^2[t_0 + \bar{s}Nt_1]. \quad (9)$$

We note that the above equation assumes network contention is negligible, which is unlikely to be true given the high volume of data. To incorporate such contention into our model would be difficult and well beyond the scope of this work. It suffices to say that communication contention will manifest itself by an apparent increase in latency and decrease in bandwidth, and is likely to be more noticeable the greater the number of processors used.

In a similar manner to that outlined above, the computation and communication times per task for the K component of the sixfold algorithm is given by

$$T^{6x}_{\text{compute}} = \bar{s}^2[N^2\beta + 2N^2n\alpha + 2Nn^2\alpha], \quad (10)$$

$$T^{6x}_{\text{comm}} = n^2[t_0 + \bar{s}^2t_1]. \quad (11)$$

In comparison to the twofold algorithm the compute time is identical, except for a factor of 2 in the integral generation component (this increases to a factor of 3 if J is also computed). The aggregate communication requirement is now, however, of $O(n^2N^2)$.

4 Parallel direct MP2

The closed-shell MP2 correlation energy is given by

$$E^{(2)} = \sum_{ijab} \frac{[ia|jb](2[ia|jb] - [ib|ja])}{\varepsilon_i + \varepsilon_j - \varepsilon_a - \varepsilon_b} \quad (12)$$

$$= \sum_{i>j} e_{ij}, \quad (13)$$

where indices i, j and a, b denote occupied and virtual canonical SCF orbitals respectively, ε_i and SCF orbital energy and e_{ij} a pair correlation energy. Only the K^{ij} operator integrals are required, and since the method is non-iterative $E^{(2)}$ can be accumulated in a piecemeal fashion by, at any time, generating K for a unique subset of (ij) pairs. This multipassing approach [3, 4] significantly extends the size of systems which can be treated within a limited amount of memory, but at the expense of repeated computation of the AO integrals. The blocking loop for K is obtained by limiting the range of the first index transformation to a subrange of the

occupied orbitals. This ensures that the N^5 work associated with the transformation remains constant, and only the N^4 work associated with integral computation increases due to multipassing. A minimum global memory requirement of $n_{occ}n_{vir}^2$ is obtained when only one occupied orbital is treated in a given pass of the integrals, in which case there are n_{occ} passes of the integrals.

Adapting the twofold or sixfold algorithms to perform paged MP2 calculations requires little extra work; the J matrix is not computed, a blocking factor is placed around the entire code to determine the number of occupied orbitals treated in each batch, and some new code is added to contract a subrange of K into the MP2 energy. In addition, however, and only for the twofold redundant algorithm, it may be advantageous to transform ζ in Fig. 1 to the virtual orbital basis prior to scattering and accumulating the partially transformed K on remote processors; since $n_{vir} < N$ this reduces the storage requirements for K and also the amount of data which is communicated. For the sixfold redundant algorithm (which actually has a fourfold redundancy when only K is computed) this is not possible.

As the global memory of a parallel machine is proportional to the number of processors used, increasing this can reduce the number of required passes over the AO integrals. This fact was earlier recognized and exploited by Márquez and Dupuis [5]. Coupled with the increased performance available from more processors this results in *super-linear* speedup, at least until multipassing of the integrals becomes unnecessary. An estimate of the twofold computation time is given by

$$t_{comp} = \frac{In_{pass} + T}{p}, \tag{14}$$

where p is the number of processors, I and T are the single-node AO integral evaluation and transformation times, respectively, and n_{pass} is the number of integral passes which in turn is a function of p, the available memory per processor and the size of K. Deviation from this estimate will be primarily due to the effects of communication and how the communication capability of the machine scales with an increasing number of processors.

5 Results and discussions

All calculations reported in this work were performed using the Intel Touchstone Delta. This machine consists of 512 i860 processors each with 16 MB of memory and linked in a 16×32 mesh topology. The standard programming model is message passing. The shared-memory facility provided by the GA tools on the Delta is implemented via interrupt handlers. A similar concept was used in early parallel CCSD [20] calculations, however, that work required very detailed programming of the interrupt handlers and was not readily portable.

As a first test case we have used butane in a cc-p VDZ basis [21] with 60 shells and, using Cartesian polarization functions, 110 basis functions, Both J and K operators were produced with the 17 occupied orbitals defining the active orbitals. With this basis and ratio of active to total orbitals, we consider this case to be somewhat representative of the transformation requirements typical of MCSCF or highly correlated calculations. The resulting elapsed times for both the twofold and sixfold algorithms are plotted as a function of the total number of processes using a $\log_{10}/\log_{10}$ scale in Fig. 3. Perfect scaling is given by a straight line of unit slope. In scaling from 8 to 256 processors, parallel efficiencies of 69% and 85% are

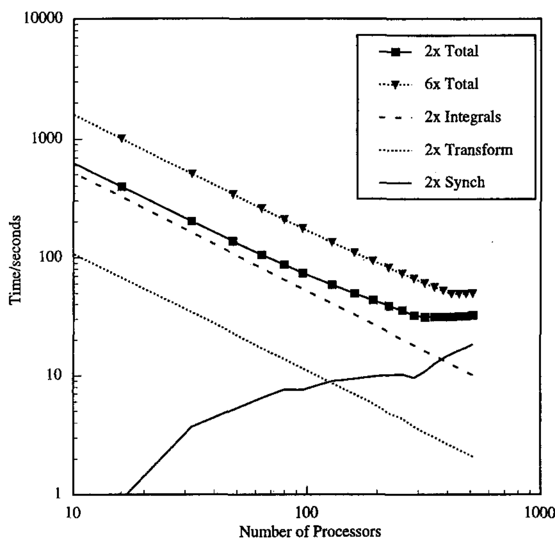

Fig. 3. Performance of twofold and sixfold algorithms for cc-pVDZ butane calculation

observed for the twofold and sixfold algorithms, respectively. Despite, however, the greater parallel efficiency of the sixfold algorithm, the twofold algorithm is always considerably faster. For comparison, on 32 processors, the parallel direct SCF calculation converged in 815 s with approximately 47 s per Fock matrix construction while the four-index transformation was completed in 201 s.

To analyze the performance further, the total elapsed time has been broken down into AO integral generation, transformation and synchronization times by placing timing calls around the relevant sections of the code. Due to dynamic task allocation, the ratio of these times will vary from processor to processor. In Fig. 3 we also plot *average* component times for the twofold butane test case. This graph shows that for low orders of parallelism, the AO integral cost is much more significant than either the transformation or communication components. It is also evident that while the integral generation and transformation times scale linearly with the number of processors, the synchronization time progressively increases. It is important to realize that linear scaling of the *average* integral generation and transformation times implies only that the corresponding total times do not increase due to some overhead associated with the parallelism. This could occur if, for example, the communications were a bottleneck. On the other hand, the increase in synchronization time implies that it is becoming progressively harder to load balance the work across the different processors. Closer inspection of this example reveals that there is a total of 1476 tasks which take, on average, 4.3 s to complete, but span a range of 1.2–29.8 s. Given the number of tasks and distribution in magnitude, it is not surprising that the effects of task granularity become manifest beyond 200 processors.

As our second test case we perform a direct MP2 energy calculation on morphine (Fig. 4) using a 6-31G basis. There were 54 correlated valence orbitals and with 101 shells and 227 functions in the AO basis. For comparison, the parallel direct SCF calculation converged in 1668 s with approximately 100 s per Fock matrix construction using 256 processors. In Table 1 we present the total elapsed time and number of AO integral passes required for a various number of processors. Comparing initially the twofold and sixfold algorithms, we find that for small

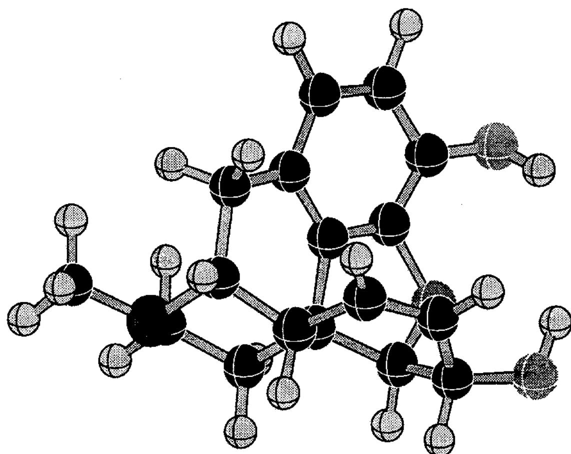

Fig. 4. Morphine ($C_{17}H_{19}No_3$)

Table 1. Multipass total times for morphine: twofold and sixfold algorithms in seconds

Nodes	Passes	Twofold	Sixfold	Nodes	Passes	Twofold	Sixfold
32	7	3656.5	7459.0	160	2	420.3	562.0
40	5	2650.9	5217.0	176	2	377.4	515.4
48	5	1994.5	3735.5	192	2	369.7	475.8
56	4	1510.9	2647.7	200	1	274.9	276.8
64	4	1329.7	2324.5	224	1	250.4	248.4
72	3	1024.4	1606.0	256	1	227.9	219.7
80	3	927.1	1461.9	288	1	203.5	196.3
96	3	780.6	1244.2	320	1	186.6	176.1
112	2	558.7	768.6	384	1	167.0	150.2
128	2	514.6	682.3	448	1	151.1	129.9
144	2	462.4	617.6	512	1	144.6	116.6

processor count the lower integral evaluation cost makes the twofold algorithm substantially faster. As the numbers of processors increase, however, the smaller communications in the sixfold algorithm yields superior performance beyond about 200 processors and by 512 processors it is some 20% faster. An estimate of the relative compute to communication ratio for the two algorithms and this test case can be obtained from the performance model outline in Eqs. (8–11). On the Intel Touchstone Delta t_0 is nominally 300 μs with the point-to-point bandwidth, t_1, equal to 1 μs per word. Empirical estimates give $\beta = 1.57 \times 10^{-5}$ s per integral and $\alpha = 7.8 \times 10^{-8}$ s per matrix multiply operation. The ratio of computation to communication is therefore approximately 2.0 and 7.7 for the twofold and sixfold algorithms, respectively.

Predicting *a priori* the crossover point where the sixfold redundant algorithm becomes more favorable is by no means simple since it is a complex function of the system being studied, the basis set used, the ratio of active to other orbitals, and the machine on which the calculation is performed. Given the results for butane and the fact that even on 512 processors the sixfold redundant algorithm is only marginally faster for morphine, we would advocate initially using the twofold

redundant algorithm on the Intel Delta. On other machines, especially those with a poorer communication network, the sixfold redundant algorithm may be more advantageous. In general, the availability of both alternatives is desirable with the user directing which algorithm is used based on previous experience.

For the morphine example a single pass of the integrals requires a minimum of 200 processors of the Delta. Using multipassing, however, the same calculation has been performed on only 32 processors and could probably be performed on 8 processors if it were not for the excessively large elapsed time. To highlight the combined effect of increased global memory and compute power obtained as processors are added we plot, in Fig. 5 and for the twofold algorithm, both linear speedup and the observed speedup relative to the 32 processor time. To comparison shows that at the maximum 512 nodes, the observed speedup is over 1.5 times better than linear. To try and isolate the effect of increased memory from that of increased compute power we have also plotted, in Fig. 5, the speedup predicted from Eq. (14) and the number of integral passes required for a given number of processors. Agreement between the model and observed results is relatively good up to about 128 processors, after which the predicted speedup becomes progressively less accurate. This, however, is to be expected given that the model ignores communications and issues associated with load imbalance. Within a region corresponding to a fixed number of integral evaluations linearity implies a scalable algorithm. This is observed, with a minor exception above about 400 processors. For optimal use of machine cycles this calculation should be run on the minimum number of processors required to allow a single pass of the integrals, viz., 200. Beyond this the algorithm can scale at best linearly, but in practice on going from 200 to 512 processors the parallel efficiencies of the twofold and sixfold algorithms are 74% and 92%, respectively.

Finally, it is instructive to note the differences between the butane and morphine test cases. The former consisted of a small number of active orbitals with a relatively large basis set with high angular momentum functions. Using 256

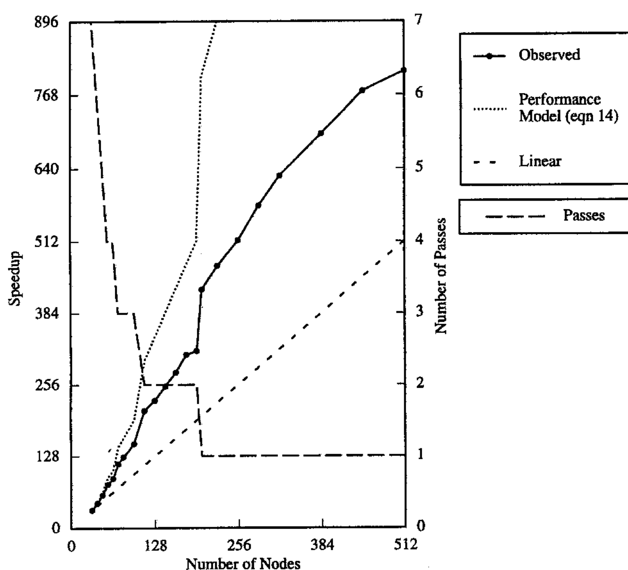

Fig. 5. Scaling of twofold MP2 algorithm for 6-31G morphine calculation, speedup relative to 32 nodes

processors, the AO integral generation required 25.9 s per node from a total time of 35.2 s. The communication cost is proportionally insignificant and the deviation from perfect scalability originates only from insufficient parallelism associated with this small test case. On the other hand, morphine with a 6-31G basis set required 53.3 s for the AO integral generation from a total of 226.9 s for the equivalent run. In this case, the volume of data communicated is comparatively large and, as a consequence, scalability is now limited by communication contention rather than task granularity.

6 Conclusions

This article addresses several issues associated with adapting a direct four-index transformation to massively parallel architectures. Two algorithms, which differ in the number of times they recompute the two-electron integrals have been devised, implemented, and tested on two quite different molecules. As expected both algorithms exhibit high levels of parallelism showing reasonable efficiency even on 512 processors. The comparative performance of the two algorithms depends greatly on the system under study, the number of processors used, and the communication capabilities of the underlying hardware. Thus, the availability of both alternatives within one package is advantageous.

The availability of a shared-memory model, and implementation of this within the Global Array tools [19] has greatly simplified the work performed here. Furthermore, these tools have allowed us to use efficiently the large global memory available on MPPs and thereby achieve superlinear speedup for the direct MP2 application. Such superlinear speedups are likely to be equally possible for other quantum chemistry applications that, like direct MP2, are driven at least in part by available memory.

Acknowledgments. This work was performed under the auspices of the High Performance Computing and Communications Program of the Office of Scientific Computing, U.S. Department of Energy under contract DE-AC6-76RLO 1830 with Battelle Memorial Institute which operates the Pacific Northwest Laboratory, a multiprogram national laboratory. This work has made extensive use of software developed by the Molecular Science Software Group of the Environmental Molecular Sciences Laboratory project managed by the Office Health and Energy Research. This research was performed in part using the Caltech Concurrent Supercomputing Facilities (CCSF) operated by Caltech on behalf of the Concurrent Supercomputing Consortium. Access to this facility was provided by Pacific Northwest Laboratory.

References

1. Taylor PR (1987) Int J Quantum Chem 31:521
2. Sæbø S, Almlöf J (1989) Chem Phys Lett 154:83
3. Head-Gordon M, Frisch MJ, Pople JA (1988) Chem Phys Lett 153:503
4. Frisch MJ, Head-Gordon M, Pople JA (1990) Chem Phys Lett 166:281
5. Márquez AM, Dupuis M (1995) J Comp Chem 16:395
6. Nielsen IMB, Seidl ET (1995) J Comp Chem 16:1301
7. Watts JD, Dupuis M (1988) J Comp Chem 9:158
8. Covick LA, Sando KM (1990) J Comp Chem 11:1151
9. Wiest R, Demuynck J, Bernard M, Rohmer MM, Ernenwien R (1991) Comp Phys Comm 62:107
10. Whiteside RA, Binkley JS, Colvin ME, Schaefer HF (1987) J Chem Phys 86:2185

11. Limaye AC, Gadre SR (1994) J Chem Phys 100:1303
12. Windus TL, Schmidt ME, Gordon MS (1994) Theor Chim Acta 89:77
13. Hampel C, Peterson K, Werner HJ (1992) Chem Phys Lett 190:1
14. Koch H, Christiansen O, Kobayashi R, Jorgensen P, Helgaker T (1994) Chem Phys Lett 228:233
15. Werner HJ, Meyer W (1980) J Chem Phys 73:2342
16. Saunders VR, van Lenthe JH (1983) Mol Phys 48:923
17. Bender C (1972) J Comp Phys 9:547
18. Nieplocha J, Harrison RJ, Littlefield RJ (1994) In Supercomputing' 94 IEEE Press, New York, p 330
19. Guest MF, Aprà E, Bernholdt DE, Fruchtl HA, Harrison RJ, Kendall RA, Kutteh RA, Nicholas JB, Nichols JA, Stave MS, Wong AT, Littlefield RJ, Nieplocha J (1994) In: Tentner AM (ed) Grand Challenges in computer simulation, high performance computing 1994, Proc. 1994 Simulation Multiconference. Society for Computer Simulation
20. Rendell A, Guest M, Kendall R (1993) J Comp Chem 14:1429
21. Dunning TH (1989) J Chem Phys 90:1007