A New Algorithm for Second Order Perturbation Theory

Andrey Asadchev Mark S. Gordon

Abstract

A new second order perturbation theory (MP2) algorithm is presented for closed shell energy evaluations. The new algorithm has a significantly lower memory footprint, a lower FLOP (floating point operations) count, and a transparent approach for the disk/distributed memory storage of the MP2 amplitudes. The algorithm works equally well on single workstations and large clusters. The new algorithm allows one to perform large calculations with thousands of basis functions in a matter of hours on a single workstation. While for most practical purposes, classical MP2 is eclipsed by density-fitting methods, the approaches and lessons learned in the presented implementation are applicable beyond the MP2 algorithm.

1 Introduction

The integral transformation, also known as 4-index tranformation, tranforms atomic (typically) integrals to molecular integrals via the simple formula:

$$(ij|kl) = C(i,p)C(j,q)C(k,r)C(l,s)(pq|rs)$$

Using the common convention, occupied indices o are designated by indices $i, j, ...$, virtual indices v are designated by indices $a, b, ...$, and atomic indices n by indices p, q, r, s.

Typically, several classes of molecular integrals are needed, eg $(ai|bj)$, $(ab|ci)$, etc. But in the case of MP2 energy,

$$t_{ij}^{ab} = \frac{2(ai|bj) - (bi|aj)}{\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b}$$

$$E_{MP2} = \sum\sum t_{ij}^{ab}(ai|bj)$$

only $(ai|bj)$ integrals are needed to form t_{ij}^{ab} MP2 amplitudes. The $(ai|bj)$ integrals and consequently t amplitude have symmetry such that $(ai|bj) = (bj|ai)$ which can be used to halve storage requirement and number of computations.

The MP2 energy calculation scales as ON^4 and requires O^2V^2 integral storage. Out of all many body methods is the cheapest and the one with lowest compute to I/O ratio.

There is a number of different algorithms developed over the years, [1, 2, 3, 4, 5, 6, 7] due to simplicity of the algorithm and its popularity. This work is to improve

the algorithm and generalize it to handle larger calculations, using either memory or filesystem as a storage medium.

2 Matrix chaining

There exists a simple matrix chaining multiplication property, which, surprisingly, is not very well-known. Given three (or more) matrices, the matrices can be multiplied without changing the outcome by two different factorizations, $A = (BC)D$ and $A = B(CD)$.

At first glance the above fact is not interesting until you consider the number of operations between the two. Suppose for example, that the dimensions are $B(k, l)$, $C(l, m)$, $D(m, n)$, and $A(k, n)$. The number of operations are $(klm + kmn)$ and $lmn + kln$ respectively.

This property can be applied to drastically reduce the number of operations in integral transformations. Suppose the integral transformation is applied in the naive order:

$$G(v, o, v, o) = C(o, n)(C(v, n)(C(o, n)(C(v, n)G(n, n, n, n))))$$

then the total number of operations is:

$$vn^4 + von^3 + v^2on^2 + v^2o^2n = vn(n^3 + on^2 + von + vo^2)$$

if the transformations are applied occupied index first, then the number of operations is:

$$on^4 + o^2n^3 + vo^2n^2 + v^2o^2n = on(n^3 + on^2 + von + v^2o)$$

and the difference between the two is on the order of a factor of v/o. Considering that typically $v > o$, the computational savings are significant.

To second benefit comes from reduced memory requirements. Since the first two transformations "shrink" atomic indices to occupied indices, the entire tensor is quickly reduced to $G(o, o, n, n)$ storage, rather than much larger $G(v, o, n, n)$ storage.

3 General Algorithm Considerations

To have a scalable algorithm, a special attention needs to be paid to memory footprint, I/O patterns, and I/O optimization by means of aggregation of smaller transfers into larger blocks.

3.1 Memory

The algorithm must have small memory print, under 1G per core on current hardware, even for large computations with several thousand basis functions. In terms of basis functions and shells, the memory overhead must be on order of M^2O^2, where M is some adjustable blocking factor, for example the size of largest shell in the basis set, otherwise any significant computation would require nodes with ten or more gigabytes of memory per *core*. For example, a computation with 3000 basis function and 300

occupied orbitals requires 22GB per *core* if memory were to scale as N^2O. The blocking factor must be adjustable to adapt to computers with different number of cores and memory.

3.2 I/O Considerations

For any significant problems size, the amplitudes are too great to store in core memory. GAMESS [8], for example has several MP2 algorithms, including two which are parallel disk-only [7] and distributed memory [5] implementations. However, using modern programming techniques, the same algorithm can be adapted to both, disk and distributed memory. The efficient access patterns between distributed memory and disk are the same: large contiguous transfers are preferred. Typically disk has much worse throughput than the distributed memory. If an algorithm works well with disk, it is guaranteed to work well with distributed memory, even when running over slow Ethernet networks.

The general efficacy for using disk is outlined by Pulay [6]. In short, smaller research groups may not have access to computers with large memory, but access to workstations with large fast disks is very common.

There is one important detail: due to buffering, writes tend to be significantly faster than reads. Therefore, algorithms which both read and write large datasets should be optimized in favor of reads.

The latency of storage access can be hidden by overlapping I/O and computations. This can be accomplished either by having a number of threads perform computations and I/O independently of one another or having a single I/O thread perform data transfers while the other thread perform computations.

Implementation transparency, e.g. distributed memory or file implementation, is easily accomplished using polymorphic interface, e.g. overriding virtual functions in C++, allowing to choose an appropriate implementation at the runtime. For example, use distributed memory if enough is available, otherwise default to filesystem backend.

3.3 File I/O considerations

There are two de facto file format and their corresponding libraries that allow easy manipulation of multidimensional scientific data on filesystem, HDF5 [9] and NetCDF[10]. For the purposes of implementing dense tensor storage, the two file format are comparable in performance and capabilities.

Storing data on the single node is straightforward. However parallel storage requires parallel file system. There are number of parallel filesystems, for example PVFS and Lustre. PVFS is easily configurable filesystem, suitable for local clusters. Lustre is more complicated filesystem, found for example on Cray supercomputers. Regardless of particular filesystem, the principle is similar to that of RAID0 [11]: an entire file is stripped over a number of I/O nodes. The performance of parallel filesystem primarily depends on the strip size and the number of I/O nodes. Both HDF5 and NetCDF have facilities for parallel I/O.

4 Naive Approach

A simple MP2 approach is described in listing 1.

The main points about the simple implementation:

- The amplitude symmetry is exploited in Q,S shells. The half transformed integrals $t2$ are writen as triangular matrices, $i <= j$ as well as its transpose $j <= i$. If running on multiple cores, each Q,S pair can be evaluated independently, allowing to benefit from overlapping computation/write.
- Integral computation and first transformation are screened using Schwarz method. Subsequent transformations are not screened.
- The matrix transformations can be done using BLAS matrix routine. Several shells can be transformed at the time to increase efficiency. The temporary memory is on the order of (O^2M^2).
- 3rd tranformation is straightforward, temporary memory required is $(O^2 * N/2)$ where N is the number of functions.
- the fourth transformation requires noncontiguous read. As mentioned above, the disk is not efficient to handle noncontiguous read. For a large problem, the 4th step becomes increasingly slow, rendering this approach extremely inefficient.

5 Better Algorithm

What is desired is an algorithm which still exploits symmetry but is somehow able to load untransformed index contiguously to maximize throughput.

If the amplitudes were stored as $t(N*N, ij)$ arrays then it would be a simple matter of reading contiguous blocks corresponding to an occupied index, transforming them, and evaluating energy, all at a cost of single read only. Keep in mind that the quantity $N * N$ is a relatively small, only 200MB for 5000 basis functions.

The problem is then how to write such data efficiently since it is generated as (ij, Q, S) shell pair at the time. Writing individual shell pairs at a time to form (QS, ij) is inefficient, for example in the case of s-shell pair, it would require a long noncontiguous write. However one can notice that to generate occupied transformation, very little memory is needed. This fact can be exploited to evaluate and to write a block of M^2 functions at a time. For example, assuming 500 occupied orbitals, working memory required is 1MB per shell function, therefore a block M^2 of 256 functions, e.g. 16 sp-shell pairs, is only 256MB but 16 separate writes can now be aggregated into a single large write.

By performing (QS, ij) and its symmetric transpose (SQ, ji) next to each other, the contigous section of write can be further doubled.

The fact that the virtual index transformation is also relatively small in terms of memory can be used to further improve I/O. If an entire *node* has 2 GB of memory, 10 (Q, S, ij) blocks can be loaded at once. This means the tensor storage can be redimensioned from $(N * N, O^2/2)$ to $(B * N * N, O^2/(2B))$, with $B = 10$ in the example, and consequently the writes can now be $(B * QS, ij/B)$, with atomic and occupied orbitals interleaved. If $B = O^2/2$ then the algorithm is essentially in-core version. The graphical depiction of the access patterns is outlined in figure 1.

```
allocate V(O*O/2,N,N); // (ia|jb) storage
for S in Shells {
  for Q <= S {
    for R in Shells {
      for P in Shells {
        // skip insignificant ints
        if (!screen(P,Q,R,S)) continue;
        t1(i,R,Q,S) = eri(P,Q,R,S)*C(i,P);
      }
      t2(i,j,Q,S) = t1(i,R,Q,S)*C(j,R);
    }
    // exploit symmetry
    V.store(t2(ij,Q,S));
    V.store(t2(ji,S,Q));
  }
}
// 3rd index
for s in N {
  t2(ij,Q) = V(ij,Q,s); // load NO^2 tile
  t3(ij,a) = t2(ij,Q)*C(a,Q); // transform
  V(ij,a,s) = t3(ij,a)); // store VO^2 tile
}
// 4th index + energy computation
for a in V {
  t3(ij,S) = V.load(ij,a,S); // load NO^2 tile
  t4(ij,b) = t3(ij,S)*C(b,S); // transform
  E += Energy(t4); // evaluate energy
}
```

Table 1: Naive approach

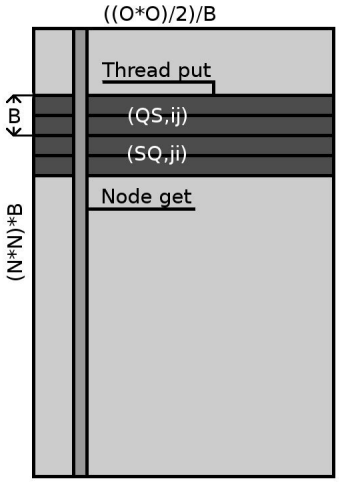

Figure 1: Amplitude access patterns

Combining the above ideas, one can develop the following algorithm, which has $(2 * M^2 * B, ij/B)$ and $(N^2 B)$ I/O respectively, Listing 2, with M and B factors determined by setting runtime memory limits.

The main points about the above implementation:

- The amplitude symmetry is exploited in Q,S shells. The half-transformed integrals are written independently and can be computed in parallel.
- The QS list is processed in terms of blocks of shell pairs, rather than individual shells pairs. The optimal block size will depend on the available memory. The bigger the block size, the better in general.
- The transformed integrals are scrambled such that shells are interleaved with blocks of ij indices of size B. The contiguous size of this noncontiguous write is $2 * M^2 * B$.
- The 3rd/4th transformation reads the contiguous interleaved blocks. The shell order is unscrambled one occupied pair at a time, the unscrambled block is transformed and the corresponding energy is computed.
- The read operation to fetch next block can overlap with computation.

The 2 innermost transformations are responsible for most of the computational work, therefore it is important to have it as efficient as possible in terms of performance and memory footprint. For any any given shell pair (q, s), the (P, R) list is evaluated in terms of blocks off identical shells, to minimize integral initialization overhead. Each individual block is contracted to first occupied index. Once a given R block is finished, it is then transformed to second occupied index. each transformation can be carried out using `dgemm`, making sure that screened out integrals are absent from transformation.

```
B = ...; // some blocking factor, according to available memory
allocate V( N*N*B, (O^2/2)/B );
for (S,Q) in Blocks(Q <= S, M) {
  // block QS pairs into blocks of M functions
  for R in Shells.blocks {
    for P in Shells.blocks {
      // skip insignificant ints
      eri.screen(P,Q,R,S));
      t_(i,S,R,Q) = eri(S,R,Q,P)*C(i,P);
      t1(i,S,Q,R) += t_(i,S,R,Q);
    }
    t2(j,i,S,Q) = t1(i,S,Q,R)*C(j,R);
  }
  t(QSB,ij/B) = t2(j,i,S,Q); // the shell order is scrambled
  V(QSB,ij/B) = t(QSB,ij/B); // write block
  V(SQB,ji/B) = t(QSB,ij/B); // and symmetrical transpose
}
// 4/3 index
for ij in (O^2/2)/B {
  t(QSB) = V(QSB,ij);
  for (i,j) in B {
    t2(Q,S) = t(QS(i,j)); // unscramble shell order
    t3(a,S) = t2(Q,S)*C(a,Q)
    t4(a,b) = t3(a,S)*C(b,S);
    E += Energy(t4);
  }
}
```

Table 2: Better approach

Table 3: Small Benchmarks, compared to DDI [5] and IMS [7].

Input	Cores	DDI	IMS	New
Taxol/6-31G	24	39.7	3.7	3.1
Taxol/6-31Gd	36	86.3	7.5	5.4

Table 4: Small Benchmarks, compared to IMS [7]

Input	6 cores	IMS/6 cores	60 cores/1GbE	60 cores/IB	IMS/60 cores
Taxol	76.0	116.6	19.3	7.9	10.7
19H20	458.3	858.5	50.5	45.7	95.0

6 Performance

There are a number of points which are useful to judge performance, scalability, and flexibility of the algorithm:

- How the new approach compares to similar algorithm
- how to network interface affects performance
- The relative time spent in ERI, transforms, and I/O

First, lets compare performance to DDI and IMS implementations in gamess to show poor performance of the former and excellent performance of the latter on an average cluster connected by InfiniBand. The two inputs are a Taxol molecule, with small 6-31G and larger 6-31Gd basis, Table 6. The DDI code is extremely slow compared to both, the IMS and the current implementation, by more than a factor of 10X. Furthermore, DDI MP2 memory requirement scales as ON^2 making it suitable only for small calculations: anything beyond 1000 atomic basis functions would require over 1GB of local memory per core, leaving little room to scale.

The next set of benchmarks to illustrate advantage of the new approach over the IMS algorithm are Taxol/cc-pVDZ and 19H20/aug-cc-pVTZ. The Taxol/cc-pVDZ computation involves 1185 basis functions, 164 occupied and 959 virtual orbitals. The 19H20/aug-cc-pVTZ computation involves 1995 basis functions, 162 occupied and 1653 virtual orbitals. The first input is less computation intensive but requires 50% more storage and consequently I/O whereas the second input is computation heavy due to diffuse (less screening) functions.

The new implementation is a clear improvement over the existing IMS disk algorithm, especially when diffuse functions are present, being faster by almost factor of two. The implementation scales on small cluster, even when running over an Ethernet interface, although more I/O bound Taxol calculation performance deteriores quickly. In case of computationally heavy water cluster input, the difference between Ethernet and InfiniBand is about 10%.

The breakdown of each step of calculation is given in Table 6. In both cases the integral calculation accounts for significant fraction of time. The water cluster calculation has almost all of its work concentrated in the integral and first transformation

Table 5: calculation breakdown

Input	Eri	T1	T2	WRITE	READ	T3+T4	sync
Taxol	38.2	28.9	6.2	0.77	0.37	23.7	1.9
19H20	39.3	45.5	6.1	0.003	0.7	4.0	4.4

Table 6: Taxol/aug-cc-pVDZ

Cores	Time	Eri	T1	T2	WRITE	READ	T3+T4	sync
512	63.9	14.1	52.8	10.0	0.1	3.5	5.5	14.0
512	52.5	17.2	53.1	15.4	0.1	0.1	8.1	6.0
1024	25.3	18.2	40.6	9.9	0.1	0.1	8.4	22.7

part due to much less screening, as opposed to sparser Taxol calculation. In both cases, the total I/O accounts for around 1% of total runtime. If the computational power were to suddenly increase, the algorithm would still be viable.

The next set of benchmarks is to illustrate capability the algorithm on a large cluster, Cray XE6. Two inputs are used, Taxol/aug-cc-pVDZ and Valinomycin/cc-PVTZ. When considering timings given below, is important to keep in mind that those numbers are for one thread only and do not give the exact nature of the system as a whole.

The smaller Taxol computation has 164 active occupied, 1659 virtual, and 2009 atomic orbitals, with 500GB integral storage. The storage is small enough to fit in distributed memory. The computation times and percentage by step is given in Table 6. The run with filesystem storage takes 17% longer, which can be expected considering the 64:1 compute to I/O ratio. When running in distributed memory entirely, the I/O overhead is hardly noticeable, due to fast Gemini interconnect. The super-linear speed-up is most likely due to cache effects of reduced memory pressure on individual nodes.

The larger computation, Valinomycin/cc-PVTZ, has 222 active occupied, 3300 virtual, and 4080 atomic orbitals. The storage required for this computation is 3.3TB requiring file storage. The amplitude file is stored on Lustre filesystem, 8 I/0 nodes, and stripping size set to 32MB. The computation times and percentage by step is given in Table 6. For this computation the I/O overhead is significant, on the order of 25%, again due to more effective screening in the absence of diffuse functions. The scalability suffers as well, both due to more I/O and unfavorable 64:1 compute to I/o ratio when running on 512 cores. Nevertheless, running the calculation that would otherwise *require* around 2000 cores should illustrate efficacy of the algorithm and flexible memory/filesystem storage.

7 GPU Implementation

There is a considerable interest in porting core quantum chemistry algorithms to GPU. Previously we were able to get moderate performance with Hartree-Fock code [**?**].

Table 7: Valinomycin/cc-pVTZ

Cores	Time	Eri	T1	T2	WRITE	READ	T3+T4	sync
256	313.8	17.8	16.4	18.0	9.3	17.5	18.2	2.8
512	204.6	7.2	20.5	16.3	18.1	7.1	28.3	2.5

However, the MP2 GPU implementation turned out to be much less successful.

The following points about innermost implementation kernels must be first highlighted:

- The integral block evaluated at once is relatively small, to keep the memory footprint low.
- The integrals are screened, therefore the coefficient matrix needs to be repacked according to block-sparse structure of the integral block.
- The first transformation is a series of relatively small matrix matrix multiplications.

While the CPU can handle the above tasks rather efficiently, the GPU runtime is inefficient at handling many small tasks, rather than few large tasks. As a result, the GPU is poorly utilized, even if using multiple streams to run several small kernels simultaneously.

The results of utilizing GPU using this particular approach a disappointing: the average performance gain was less than 10% over a single CPU core. Although the overall performance of the algorithm is superior, especially over DDI algorithm, the main contribution is due to better algorithm implementation itself rather than the raw performance of the GPU.

The only place where GPU math libraries could make a difference are the last two transformations where the bulk of work is handled by two large consecutive matrix multiplies. However they don't account for much of the runtime, 30 % at most in the above examples. Speeding up those alone computations is unlikely to improve general case performance significantly.

The above finding does not mean an efficient MP2 GPU algorithm is not possible. However, to achieve good GPU utilization, an approach significantly different from the above is needed. This is in contrast from RI-MP2 GPU implementations [**?**] where the bulk of work is handled by few large matrix multiplies without the need to accomodate the integral sparcity directly.

8 Conclusions

The work described in this paper offers an improvement over the existing MP2 energy algorithms both in terms of execution time and resources utilization. A flexible data storage model allows to transparently use either filesystem or distributed memory to store partially transformed integrals. A number of sample calculations showed implementation to work well with small clusters and scale into thousands of cores on Cray supercomputer. However, translating the CPU approach into GPU implementation

proved to be unsuccessful, since the GPU runtime handles the workload composed of large number of small computations poorly.

References

[1] M. Head-Gordon, J.A. Pople, and M.J. Frisch. Mp2 energy evaluation by direct methods. *Chemical physics letters*, 153(6):503–506, 1988.

[2] M.J. Frisch, M. Head-Gordon, and J.A. Pople. Semi-direct algorithms for the mp2 energy and gradient. *Chemical physics letters*, 166(3):281–289, 1990.

[3] A.T. Wong, R.J. Harrison, and A.P. Rendell. Parallel direct four-index transformations. *Theoretical Chemistry Accounts: Theory, Computation, and Modeling (Theoretica Chimica Acta)*, 93(6):317–331, 1996.

[4] M. Schütz and R. Lindh. An integral direct, distributed-data, parallel mp2 algorithm. *Theoretical Chemistry Accounts: Theory, Computation, and Modeling (Theoretica Chimica Acta)*, 95(1):13–34, 1997.

[5] B.G.D. Fletcher, P.R. Alistair, and P. Sherwood. A parallel second-order møller-plesset gradient. *Molecular Physics*, 91(3):431–438, 1997.

[6] A.R. Ford, T. Janowski, and P. Pulay. Array files for computational chemistry: Mp2 energies. *Journal of Computational Chemistry*, 28(7):1215–1220, 2007.

[7] K. Ishimura, P. Pulay, and S. Nagase. A new parallel algorithm of mp2 energy calculations. *Journal of computational chemistry*, 27(4):407–413, 2006.

[8] M. S. Gordon and M. W. Schmidt. *Advances in electronic structure theory: GAMESS a decade later*, pages 1167–1189. Elsevier, Amsterdam, 2005.

[9] The hdf group. hierarchical data format, version 5. http://www.hdfgroup.org/HDF5.

[10] Network common data form. http://www.unidata.ucar.edu/software/netcdf/.

[11] D.A. Patterson, G. Gibson, and R.H. Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.

References

[1] M. Head-Gordon, J.A. Pople, and M.J. Frisch. Mp2 energy evaluation by direct methods. *Chemical physics letters*, 153(6):503–506, 1988.

[2] M.J. Frisch, M. Head-Gordon, and J.A. Pople. Semi-direct algorithms for the mp2 energy and gradient. *Chemical physics letters*, 166(3):281–289, 1990.

[3] A.T. Wong, R.J. Harrison, and A.P. Rendell. Parallel direct four-index transformations. *Theoretical Chemistry Accounts: Theory, Computation, and Modeling (Theoretica Chimica Acta)*, 93(6):317–331, 1996.

[4] M. Schütz and R. Lindh. An integral direct, distributed-data, parallel mp2 algorithm. *Theoretical Chemistry Accounts: Theory, Computation, and Modeling (Theoretica Chimica Acta)*, 95(1):13–34, 1997.

[5] B.G.D. Fletcher, P.R. Alistair, and P. Sherwood. A parallel second-order møller-plesset gradient. *Molecular Physics*, 91(3):431–438, 1997.

[6] A.R. Ford, T. Janowski, and P. Pulay. Array files for computational chemistry: Mp2 energies. *Journal of Computational Chemistry*, 28(7):1215–1220, 2007.

[7] K. Ishimura, P. Pulay, and S. Nagase. A new parallel algorithm of mp2 energy calculations. *Journal of computational chemistry*, 27(4):407–413, 2006.

[8] M. S. Gordon and M. W. Schmidt. *Advances in electronic structure theory: GAMESS a decade later*, pages 1167–1189. Elsevier, Amsterdam, 2005.

[9] The hdf group. hierarchical data format, version 5. http://www.hdfgroup.org/HDF5.

[10] Network common data form. http://www.unidata.ucar.edu/software/netcdf/.

[11] D.A. Patterson, G. Gibson, and R.H. Katz. *A case for redundant arrays of inexpensive disks (RAID)*, volume 17. ACM, 1988.