New efficient large-scale fully asynchronous parallel algorithm for calculation of canonical MP2 energies.

Alex A. Granovsky

Laboratory of Chemical Cybernetics, M.V. Lomonosov Moscow State University, Moscow, Russia

May 10, 2003

Outline

- Preface
- Serial MP2 energy code
- Parallelization problems
- P2P communication interface
- Parallel MP2 energy code
- Sample applications

Notation

- N - number of AO basis functions
- c - number of core orbitals
- n - number of occupied orbitals, $n \ll N$
- V - number of virtual orbitals
- $V = N - n - c \approx N$
- $i,j \in n$
- $a,b \in V$
- $p,q,r,s \in N$

MP2 energy correction formula

$$E_{corr} = \frac{1}{4} \sum_{ijab} \frac{\left((ia|jb) - (ib|ja) \right)^2}{E_i + E_j - E_a - E_b}$$

- Basically, the problem of integral transformation.
- Only integrals of the internal exchange type are required.
- To evaluate energy, it is necessary to have both (ia|jb) and (ib|ja) integrals at the same time.

Existing approaches

- Memory requirements:
  - N^4 (Saebo, Almlof 1989)
  - N^3 (Head-Gordon et al. 1988, Dupuis et al. 1994)
  - **N^2** (PC GAMESS 1997, Pulay et al. 2001)
- AO integrals permutation symmetry usage:
  - Eightfold (Dupuis et al. 1994, Schutz et al. 1999)
  - Fourfold (Head-Gordon et al. 1988)
  - **Twofold** (Saebo, Almlof 1989)
- Parallelization:
  - Over one index (Dupuis et al. 1994, Nielsen, Seidl 1995)
  - Over index pairs (Nielsen, Seidl 1995, Baker, Pulay 2002, **present work**)

Our goals

- Large systems => N^2 memory.
- Minimal number of floating point (FP) operations.
- Extensive use of the efficient matrix-matrix multiplication routine (dgemm).
- Efficient use of disk I/O and minimal sorting overhead, good CPU usage.
- Scalability and high degree of parallelization => parallel over index pairs.

Serial MP2 energy code

Two pass MP2 energy code in the PC GAMESS - main features

- Has only $O(N^2)$ memory requirements.
- Direct in the sense that AO integrals are evaluated as needed.
- AO integrals are always recomputed four times.
- Uses disk storage: $O(n^2N^2)$ disk space required.
- Uses minimal number of FP operations.

How it works:

- First pass: (pq|rs) -> (iq|js) half-transformation is performed for all fixed qs pairs and all i, j. Presorted buffers are saved to disk in direct access file (DAF) using sequential writes and in memory control structures (lists of records).
- Second pass: presorted half-transformed integrals are fetched from the disk buffers, then the second half-transformation is performed: (iq|js) -> (ia|jb) for all fixed ij pairs and all q, s. MP2 energy correction is accumulated.

First pass:

```
Initialize memory presort structures
Loop over all shells $q_{sh}$ (q loop)
    Loop over shells $s_{sh} \leq q_{sh}$ (s loop)
        ✦ for all q in $q_{sh}$ and s in $s_{sh}$ compute (pq|rs) (all p, r) using abelian symmetry
        ✦ store them in the (small) temporary file if no memory available to handle all of them
        ✦ perform first one-index transformation: (pq|rs) -> (iq|rs). Two possible ways: either using dgemm (dense case) or using daxpy (sparse case or high symmetry)
        ✦ perform second one-index transformation: (iq|rs) -> (iq|js) for all i, j using dgemm
        ✦ put (iq|js) and (js|iq) ($i \geq j$, all q and s) into presort buffers in memory
        ✦ write filled presort buffers to disk, update lists of records numbers
    End loop over s shells
End loop over q shells
Flush presort buffers to disk
```

One-index transformation

- basically a sequence of matrix-matrix multiplications:

$$(iq|rs) = \sum_p (pq|rs) \cdot C_{pi}$$

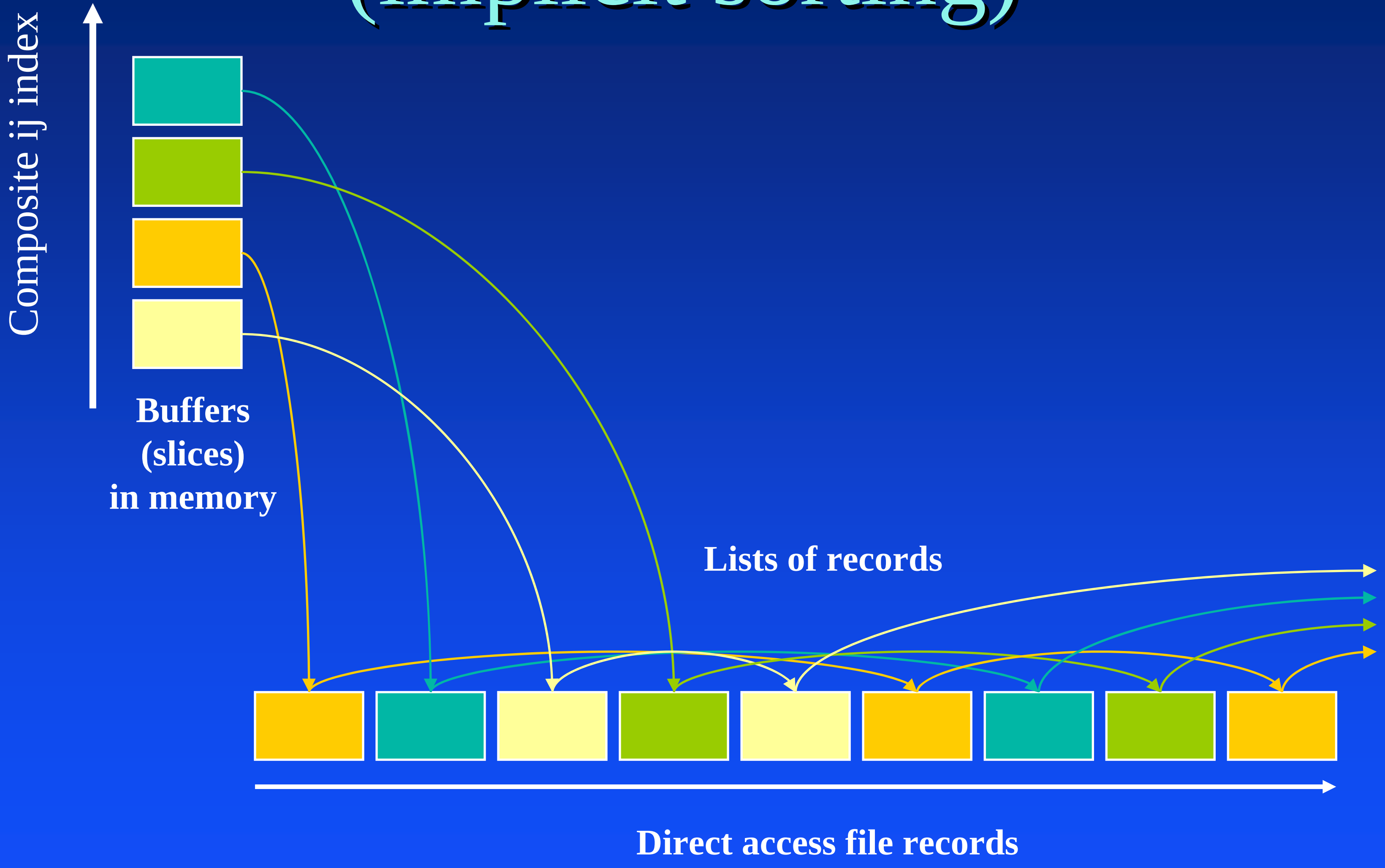
First one-index transformation

- Two ways: dense and sparse case:
  - Dense case:
    - more memory is required to hold square **N by N** matrices of AO integrals.
    - matrix-matrix multiplication uses dgemm and is very efficient.
    - transparently multithreaded via MKL SMP support.
  - Sparse case:
    - Less memory is required - only **N by n** matrices are used to hold quarter-transformed integrals.
    - AO integrals are processed on the fly using daxpy.
    - Less efficient due to use of daxpy instead of dgemm => the degree of sparsity should be high enough to compensate.
    - More problems with SMP support.

Data presorting

- After first half-transformation, we have (iq|js) integrals for all i, j and fixed qs pairs. For the second half-transformation we need them for a fixed ij pairs, and all q, s. Thus the transposition of the four-index array is required for the second pass.
- The PC GAMESS code uses efficient technique to perform this data reordering implicitly during first pass.

Modified Yoshimine sorting (implicit sorting)



Second pass:

```
Loop over ij slices
    Loop over disk buffers from the corresponding list
        ✦ read buffer into memory and fill matrices $C^{(ij)} = (iq|js)$
    End loop over disk buffers
    Loop over ij in slice
        ✦ transform (iq|js) to (ia|jb) using dgemm & symmetry
        ✦ accumulate $E_{MP2}$
    End loop over ij in slice
End loop over ij slices
```

MP2, UMP2, and ROHF-MBPT2 cases

- The implementations are very close
- MP2 and UMP2 share the first pass, in the latter case n is simply $n_\alpha + n_\beta$
- The second pass is identical for UMP2 and ROHF-MBPT2 and differs slightly from RHF case
- In the case of ROHF-MBPT2, the first pass can be either UHF-like or can be based on the more advanced approach.

Luciferine TZV* example, 398 b.f., 26 core, 46 occupied orbitals; running on AMD Athlon 1200 MHz system

	GAMESS DDI	GAMESS direct	PC GAMESS direct	PC GAMESS conventional	PC GAMESS-specific code
Minimum memory required	5MW(C)+173MW(S)	33MW	33MW	33MW	2.5MW
Memory used	5MW(C)+173MW(S)	159MW	159MW	159MW	4MW
CPU time, sec	17074(C)+2252(S)	18857	9894	5068	2812
Total time, sec	19348	18858	9895	5652	2913

Example analysis

- The DDI-based method is the slowest.
- The direct method in the PC GAMESS is approximately 2 times faster than that of original GAMESS - this is the result of Intel-specific optimization.
- The conventional method is approximately 2 times faster than direct due to integral packing.
- The PC GAMESS specific method is the fastest.
- It requires very small amount of memory.
- It has very good CPU usage.

Parallelization problems

Parallel code goals

- On the first pass, different qs pairs should be distributed over nodes.
- On the second pass, different ij pairs should be distributed over nodes.

Main problem

- During first pass, each node will produce (iq|js) for the subset of q,s pairs and all i,j.
- During second pass, each node will need (iq|js) for the subset of i,j and all q,s pairs.

$$\Downarrow$$

- Each node should communicate with all other nodes to send and receive the data the node will need for the second pass (parallel sorting is required).

Parallel sorting

- Explicit sorting stage should be avoided to improve performance.
- Parallel implicit sorting combined with the first pass is required to sort & properly distribute $O(n^2N^2)$ half-transformed integrals over nodes.

Parallel sorting problem

- Node X will send data to node Y when they will be computed by the node X => node X can use synchronous send because it knows when to send.
- Node X will receive data at unpredictable moments when they will be computed by other nodes and then delivered via inter-connection network => node X cannot use synchronous receive because it does not know when to receive.

Parallel sorting problem

- It seems that problem cannot be solved using standard interfaces like MPI

⇓

- Parallel MP2 code requires dedicated execution model and communication interface - *point to point (P2P) interface*.

P2P communication interface

P2P interface philosophy

- Point-to-point message oriented interface
- Logically separates calculations and processing of incoming/generation of outgoing messages
- Fully asynchronous very fast background processing of incoming messages using user-provided callback routines
- Fully asynchronous very fast background generation of outgoing messages, if required

Parallel sorting solution

- Each node should use asynchronous receive routine

↓

- It is necessary to use multithreaded code
- At least two threads are required - first for computations & sends (worker) and second to receive data (P2P receiver thread)

P2P execution model

- Initialization over MPI.
- Fully connected topology - each node connected to all other nodes via TCP.
- Dedicated high-priority P2P receiver thread(s).
- Receiver thread calls user callback function to process each incoming message (and send response messages if necessary).
- Thread-safe implementation. In particular, user callback can call P2P functions as well.

Basic P2P interfaces

- *P2P_Init(int nnodes, int mynode, int maxmsgsize)*
- *P2P_Sendmessage(int dstnode, int msglen, int   *message)*
- *P2P_Setcallback(P2PCB callback)*
- *P2P_Shutdown(void)*

P2P vs. other interfaces

- The real power of P2P is the user callback function called from a separate thread which has access to all node’s resources including memory and files.
- GAMESS’ (US) DDI interface can be easily (and more efficiently) emulated using P2P.
- MPI can be emulated over P2P but it is much simpler to use MPI + P2P combo.

Dynamic load balancing over P2P

- Implementation of DLB over MPI is not a trivial problem
- Implementation of global shared counters requires less than 20 lines of code using P2P
- High-performance *asynchronous* DLB

Parallel MP2 energy code

The most elegant solution

- Use of P2P asynchronous receive feature with user-provided callback routine to process incoming data (sort them and write to the disk in the case of MP2 sorting).

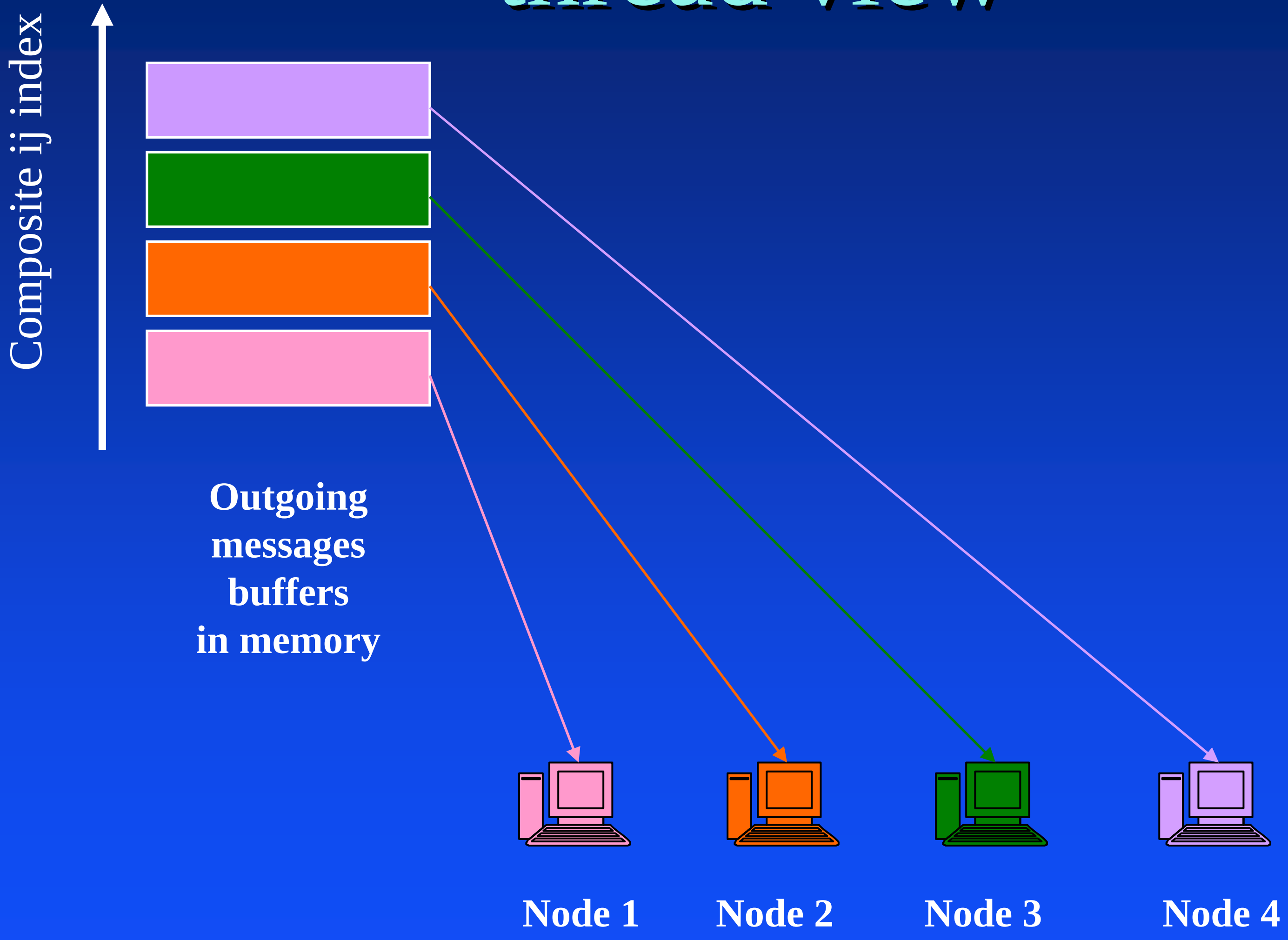
Main features

- Basically the same as for sequential code.
- On the first pass, different qs pairs are distributed over nodes (N_{nodes} total) either statically or dynamically.
- Implicit parallel sorting is performed during first pass. $O(n^2N^2)/N_{nodes}$ disk space is required on each node to store half-transformed integrals.
- On the second pass, different ij pairs are statically distributed over nodes.

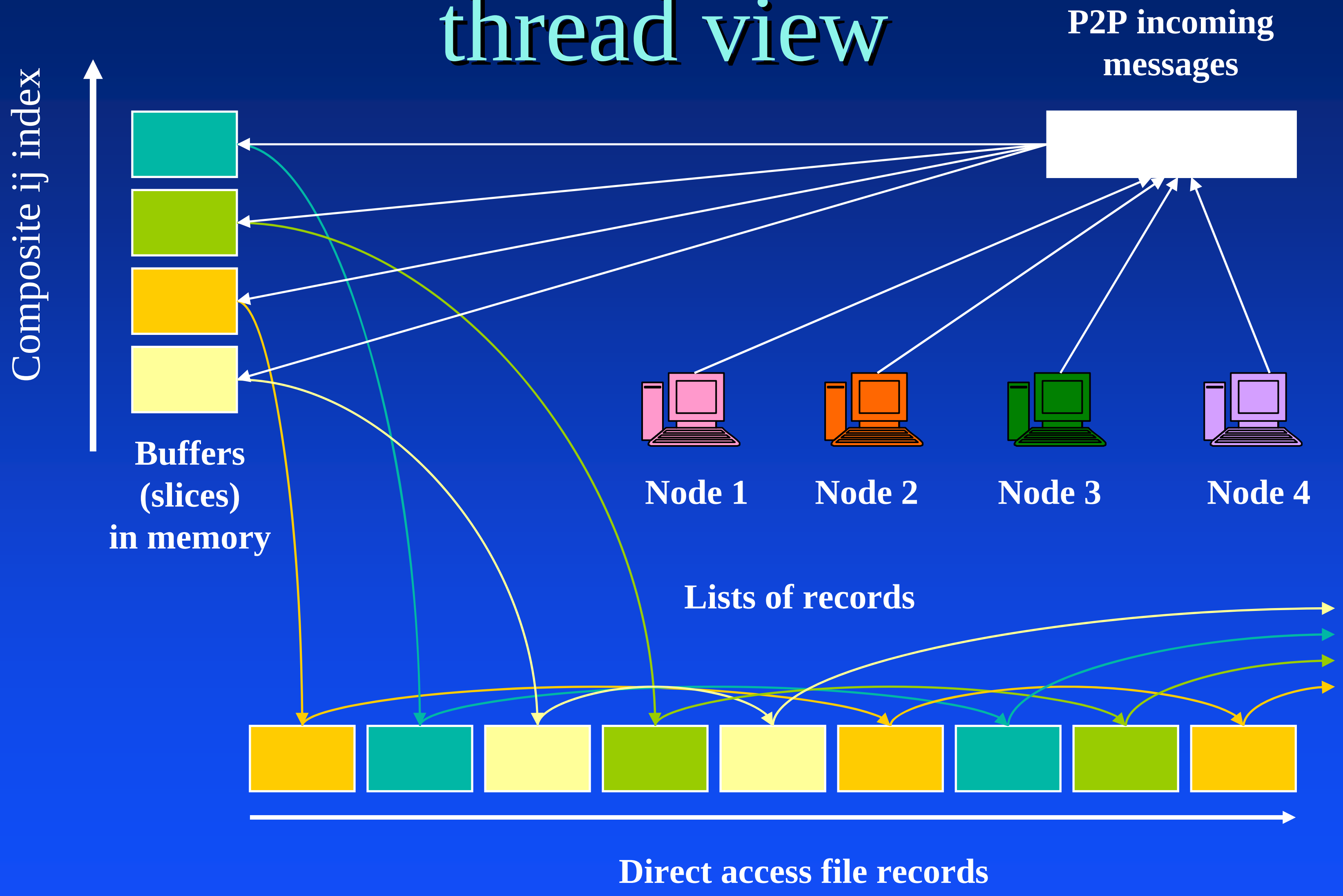
How it works:

- First pass:
  - (pq|rs) -> (iq|js) half-transformation is performed for the assigned (or dynamic) subset of fixed qs pairs and all i, j on all nodes. The half-transformed integrals are sent to the corresponding nodes.
  - At the same time, on each node, P2P thread receives the subset of half-transformed integrals from all nodes and puts them into presort buffers. Presorted buffers are saved to disk in DAF using sequential writes and in memory control structures (lists of records).

Parallel implicit sorting, worker thread view



Parallel implicit sorting, P2P thread view



How it works:

- <u>Second pass</u>:
  - presorted half-transformed integrals are fetched from the disk buffers, then the second half-transformation is performed on each node: (iq|js) -> (ia|jb) for the assigned subset of fixed ij pairs and all q, s. MP2 energy correction is accumulated.
- <u>Finally:</u>
  - the global summation of the partial contributions to the MP2 energy from each node is performed.

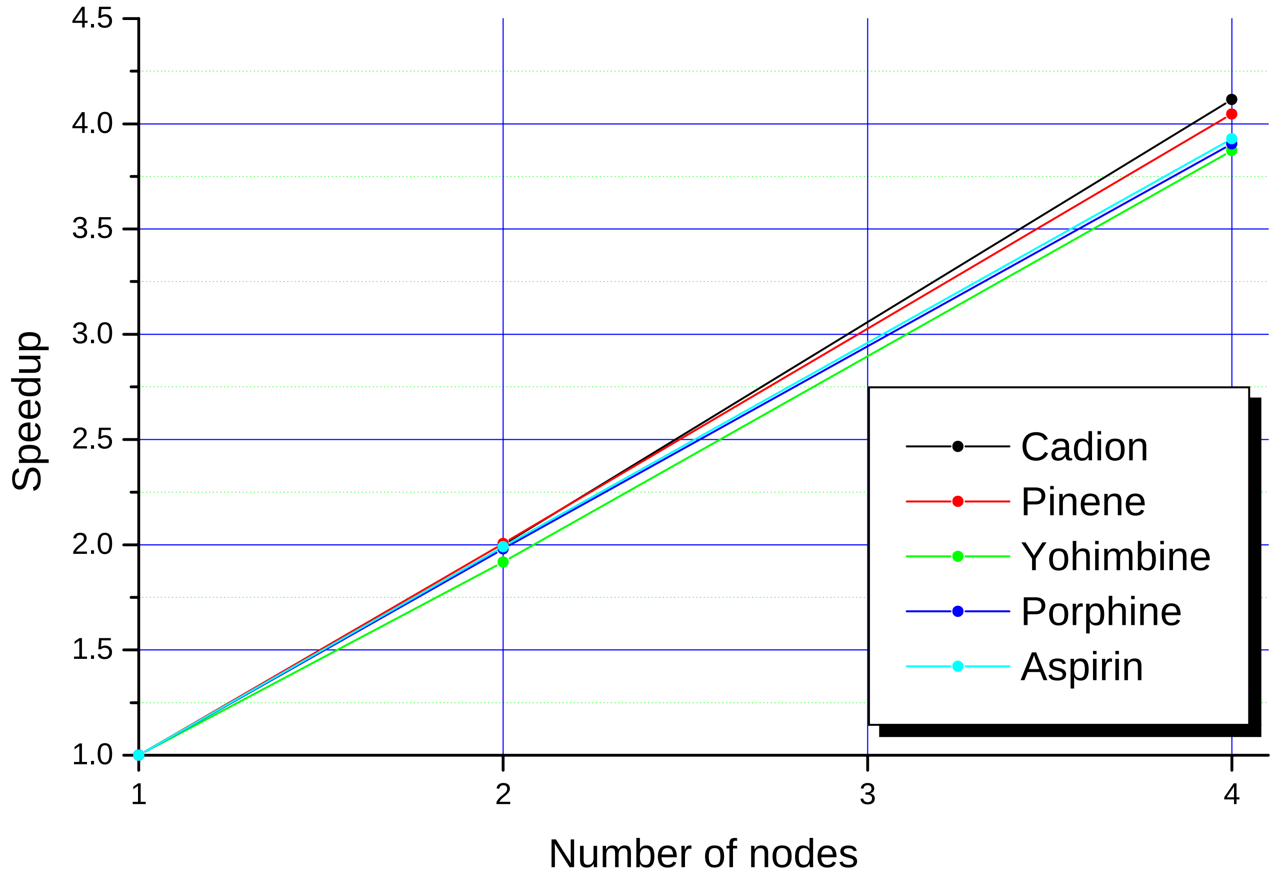
Sample applications

Small molecules

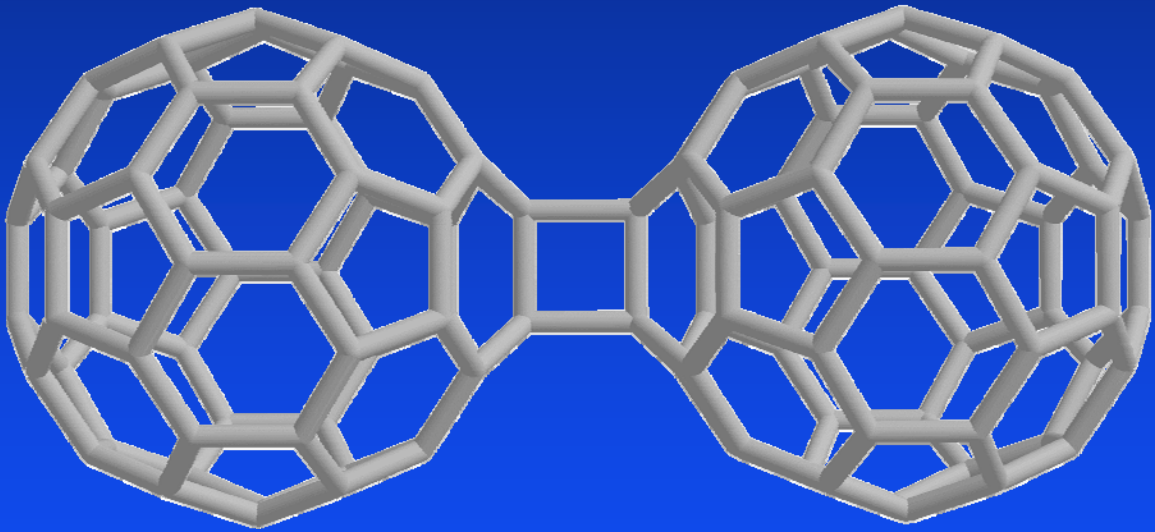
System		*Aspirin*	*Porphine*	*Yohimbine*	*α-Pinene*	*Cadion*
Basis		**6-311G****	**6-31G****	**6-31G****	**6-311G (3df,3p)**	**cc-pVTZ**
N		**295**	**430**	**520**	**602**	**1120**
c		**13**	**24**	**26**	**10**	**26**
n		**34**	**57**	**69**	**28**	**64**
Total time, min.	**1 node**	**37**	**47**	**282**	**646**	**-**
	2 nodes	**19**	**24**	**147**	**322**	**1579**
	4 nodes	**10**	**12**	**73**	**160**	**767**

Pentium III Xeon (1024KB) 500MHz / 512MB / 25GB / Fast Ethernet

Small molecules



Large molecules: Fullerene C_{60} and its dimer C_{120}



Large molecules

System	C_{120}	C_{60}
Basis	cc-pVDZ	cc-pVTZ
Group	D_{2h}	D_{2h}
N	1800	2100
c	120	60
n	240	120
N_{nodes}	20	19
Distributed DAF size	771 GB	347 GB
2e integrals calculation time, sec.	3300*4	5505*4
Total first pass CPU time, sec.	66647	57133
Second pass CPU time, sec.	36962	17937
Total CPU time per node, sec	103680	75181
Wall clock time, sec.	112697	79490
CPU usage, %	92	94.58
Node performance, MFlop/s	330	295
Performance, % of peak	66	59

Pentium III 500MHz / 512MB / 55GB / Fast Ethernet

Largest MP2 calculation attempted so far

System	C_{120}		
Basis	cc-pVTZ-f		
Group	D_{2h}		
N	3000		
c	120		
n	240		
N_{nodes}	18		
Dynamic load balancing	off	on	on
Real time data packing	off	on	on
Asynchronous I/O	off	off	on
Total FP operations count	$3.24 \cdot 10^{15}$	$3.32 \cdot 10^{15}$	$3.32 \cdot 10^{15}$
Distributed data size	2.0 TB	2.0 TB	2.0 TB
CPU time on master node, sec	83029	89301	95617
Wall clock time, sec.	150880	110826	95130
CPU usage, %	55	80.5	100.5
Node performance, MFlops/s	1330	1935	2320
Performance, % of peak	27.7	40.3	48.3
Cluster performance, GFlops/s	23.9	34.8	41.7

Pentium 4C with HTT 2.4 GHz / 1024MB / 120GB / Gigabit Ethernet

Thank you for your attention!