

Computational Chemistry Challenges: Application and Tuning of *Gaussian*® 98 on UltraSPARC Systems

Alistair P Rendell

Supercomputer Facility
Australian National University
Canberra, ACT0200
Australia

Barbara Perz

HPC Customer Benchmarking
Sun Microsystems, Inc.
Beaverton, Oregon USA

Abstract

In January 1999 the Australian National University (ANU) and Sun Microsystems, Inc. began a joint project aimed at benchmarking and improving the performance of the *Gaussian*® 98 quantum chemistry program for Sun UltraSPARC platforms. This paper reports progress made as a result of this project. The paper is broken down into three sections. In section 1 we give a brief introduction to computational chemistry and highlight some of the computational chemistry challenges being tackled at the ANU. In section 2 we provide an overview of *Gaussian*® 98, our work to improve the performance of the code on UltraSPARC systems, why this code can also require large memory and disk space, and the general structure and state of parallelism in the Solaris version of the code. Finally in section 3 we give an overview of the current performance of *Gaussian*® 98 on a variety of UltraSPARC systems.

1. Computational Chemistry Challenges

Chemistry is the study of atoms and molecules and how they interact, and *computational* chemistry aims to do this using computers. What sort of problems are tackled in computational chemistry varies hugely, but may for example involve calculating accurately the energy dissipated when methane is burnt in air, or trying to understand and predict the way large proteins containing tens of thousands of atoms interact.

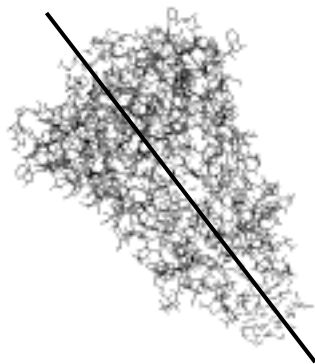
Exactly how atoms and molecules interact is determined by quantum mechanics, the underlying equations of which were derived well over fifty years ago. The problem is, however, that for all but a few very simple systems it is impossible to solve these equations analytically. Hence, by necessity, we are forced first to make approximations and then to resort to numerical computation. Not surprisingly the fewer the number of approximations made the greater the computational effort required, and the dilemma for every computational chemist is to weigh required accuracy with available computing resources. Herein lies one of the reasons why computational chemists have acquired the reputation for an insatiable demand for increased computing power.

As an illustration we consider briefly two “computational chemistry challenges” being tackled at the ANU. In the first we are interested in the properties of so called “ion channels”, while in the second we are looking at one of the basic reactions that underpins photosynthesis.

1.1. The Potassium Ion Channel

Ion channels are found in biological cell walls and are responsible for allowing the transfer of certain types of charged atoms or ions across cell membranes, while preventing others from entering the cell. Recently the first structure of an ion channel was experimentally determined [1]. This channel (Figure 1) permits the passage of potassium ions, but essentially excludes the passage of the very similar sodium ion. To understand how the selectivity of this channel works is of fundamental interest to the biochemical community.

Figure 1: The potassium ion channel with line showing central channel



The potassium ion channel contains on the order of six thousand atoms, but if we include parts of the surrounding cell membrane and then add water molecules the number of atoms quickly increases. Our studies are looking at this system at two different levels. In the first instance we are interested in the dynamical properties of the system and modeling the path taken by the potassium ion through the channel. To do this we are using classical Molecular Dynamics (MD). In this method the forces between all pairs of atoms are crudely approximated, the total force on each atom is obtained as a sum of all the pair forces, and the motions of the atoms is determined by solving Newton’s equations of motion (i.e. numerically integrating an ordinary differential equation). In the second instance we fix the structure of the entire system and are performing approximate Quantum Mechanical (semi-empirical QM) calculations to determine the electrostatic potential along the center of the channel. The major difference between the two studies lies in the number of individual calculations that must be performed. In the MD work we evaluate the forces on all the atoms at an interval that corresponds to just 10^{-15} of a second, and then run the simulation for a total real time of about 10^{-9} seconds. To do this requires of the order of 10^6 force calculations, so to complete this in a reasonable elapsed time necessitates that each calculation only takes a few seconds of computer time. Conversely, when evaluating the electrostatic potential we are interested in just a few structures and can afford to spend several hours or indeed days computing each.

1.2. The Photosynthetic Reaction Center

In the previous example the fundamental scaling of the MD and semi-empirical QM methods is linear with system size [2], i.e. the computer time and memory requirement increases linearly with the number of atoms in the system. The accuracy of both methods is, however, limited and to obtain, for example, reaction energies or vibrational properties that can rival those obtained from experiment we must move to more accurate “*ab-initio*” QM methods. Although there is much active work in producing linear scaling algorithms for these methods [3], such schemes are in their infancy, and even then it is likely that both the prefactor and the system size required before linear scaling begins to take effect are large. Thus currently there is a variety of different *ab-initio* QM methods available with a range of accuracy and associated computational cost. Typically the cost ranges from “cheap” linear scaling methods to very expensive methods that scale with the seventh power of the number of atoms in the system.

In recent work we have been using *ab-initio* QM methods to study the structure, function and spectroscopy of bacterial photosynthetic reactions centers. These reaction centers are found in purple bacteria and convert solar energy into chemical energy. In this respect they are similar to plant photosynthetic reaction centers, but have the advantage of being much simpler and as a result are better understood. In the core process light energy is funneled by a complex series of antenna to a dimer of bacteriochlorophyll, which causes it to ionize (lose an electron) creating a charged dimer cation radical. Our interest is to understand the degree to which the positive charge on the dimer cation is delocalised, and to this end the structure, and vibrational frequencies for the neutral dimer, the charged dimer cation, and the first electronic excited state of the charged cation are of interest.

Our model dimer system is shown in Figure 2; it contains a total of just 130 atoms. We have effectively performed just three calculations, and even then we were forced to use the relatively cheap density functional QM method with a very small basis set. The calculations were performed using the *Gaussian*® 98 program suite [4]. Each calculation took about 1 week of elapsed time running on 8 processors of our Sun E3500 400MHz, and used 2GBytes of memory and about 14GBytes of disk. Such calculations are at the forefront of what we can currently perform, but are only at the start of what we would really like to begin to study using *ab-initio* QM methods. With this in mind we have been working to improve the performance of *Gaussian*® 98



Figure 2: Model dimer of bacteriochlorophyll

2. *Gaussian*® 98 Overview

The *Gaussian*® 98 program suite [4] is arguably the most widely used computational chemistry package. Indeed it has been estimated that over 90% of all quantum chemical calculations are performed using the *Gaussian*® series of programs [5]. The current release, *Gaussian*® 98, appeared in the latter half of 1998 and represents a significant enhancement over the previous release (*Gaussian*® 94). The new program includes a wide range of functionality, from simple molecular mechanics through semi-empirical QM methods to advanced *ab initio* techniques (both density-functional-theory (DFT) and Hartree–Fock (HF) based). For most methods, analytic first and second derivatives of the energy with respect to nuclear displacements are available (denoted E' and E'' respectively), and these can be used with a variety of algorithms for exploring potential-energy surfaces. The new version of *Gaussian*® also includes so-called hybrid methods that use different levels of theory for different parts of the system, and linear-scaling semi-empirical and DFT methods.

Running a *Gaussian*® calculation involves running a series of separate executables or 'links'. In general, which links are run is determined automatically based on the type of calculation to be performed. Before terminating, and if required, each link initiates the next link in the series. Communication between links is performed by storing data to and retrieving data from disk files.

Gaussian® 98 is very large with over seven hundred thousand lines of code producing more than seventy links. The majority of the code is written in Fortran 77, although C is used for system related operations like memory allocation, initiating links, and input/output. Versions of *Gaussian*® 98 are denoted by *letter.number*, where *letter* is A,B,C... and *number* is 1,2,3..., and changes in the *letter* represent major changes in functionality, while changes in the *number* represents minor enhancements/bug fixes/newly supported platforms. At the time of this writing the current version is A.7.

2.1. *Gaussian*® 98 Tuning

Our initial tuning work has concentrated on HF and DFT methods, and energy, gradient and frequency calculations. The starting point for our work was *Gaussian*® 98 release A.6. Our results to date have:

- Recommended that the code be compiled using the new Sun Performance Workshop 5.0 compilers with a modified list of compiler options.
- Provide changes to the source to enable it to set buffer sizes according to the cache size of the machine being used
- Provide changes to enable file sizes in excess of 2GBytes to be supported (current limit is 16GBytes)

These changes have been included in *Gaussian*® 98 A.7.

To illustrate the effects of these changes we give in Table 1 specifications for a range of benchmark calculations covering most of the commonly used *Gaussian*® 98 *ab-initio* functionality, and in Table 2 the CPU times required to run these benchmarks using the A.6 and A.7 versions of the code. The results show that in some cases a speedup of 2.8 has been obtained.

Table 1: Benchmark Specifications

No.	Name	Formula	Theory	Cal.	Atoms/Sym.	Basis/Functions
Test178						
1	e178hf	C6H6N6O6	HF	E	24/D3H	6-31G**/300
Test397						
2	g397b3	C54H90N6O18	B3LYP	E'	168/C1	3-21g/882
Alpha-pinene						
3	ealpb3	C10H16	B3LYP	E	26/C1	6-311++G(3df,3p)/598
4	falpb3	C10H16	B3LYP	E''	26/C1	6-31G**/182
5	falpmp	C10H16	MP2	E''	26/C1	6-31g**/182
Isobutene						
6	eisoqc	C4H8	QCISD(T)	E	12/C2V	6-311++G**/148
C20H42						
7	ec20mp	C20H42	MP2	E	62/C2H	6-31g*/384
8	gc20mp	C20H42	MP2	E'	62/C2H	6-31g*/384
Benzene						
9	ebenca	C6H6	CASSCF	E	12/nosym	6-31+g(3df)/240
Acetyl-phenol						
10	gaphcs	C8H8O2	CIS	E'	18/C1	6-31++G/154

Table 2: Performance of *Gaussian*® 98 version A.6 and A.7 on one 400MHz UltraSPARC processor

Benchmark	A.6 CPU Sec	A.7 CPU Sec	A.7/A.6
test178	273	177	1.5
test397	151966	59215	2.6
ealpb3	123971	44466	2.8
falpb3	40889	19954	2.0
falpmp	115701	79563	1.5
eisoqc	3928	3348	1.2
ec20mp	3869	2449	1.6
gaphci	7612	2827	2.7
ebenca	1294	1113	1.2

The tuning change that probably has the largest effect on overall performance is the setting of buffer sizes according to the processor cache size. To understand what is done and why requires some knowledge of the way the *Gaussian*® 98 code works. In all of the benchmarks a given number of gaussian basis functions are placed on each atom in the molecule. For example atom *A* will have a number of functions of the following form:

$$\phi_i = x_A^l y_A^m z_A^n \exp(-\alpha r_A^2) \quad (1)$$

where *x*, *y* and *z* are cartesian coordinates with respect to atom *A*, *l*, *m*, *n* are integers, α is input (indirectly usually) by the user, and *r* is the distance from atom *A*. The total number of these

functions is given in the last column of Table 1, e.g. for the e178hf benchmark there are 300 gaussian basis functions.

For all the benchmarks given in Table 1 the time dominant parts of the calculations involve operations associated with these basis functions. For example in the e178hf benchmark the time dominant process is construction of the ‘‘Fock’’ matrix (F_{ij}):

$$F_{ij} = h_{ij} + \sum_{kl}^N ([ij|kl] - 1/2[ik|jl])D_{kl} \quad (2)$$

and in particular the generation and processing of the quantities in square brackets, which are integrals over the basis functions:

$$[ij|kl] = \iint \phi_i^*(1)\phi_j^*(1)|1/r_{12}|\phi_k(2)\phi_l(2) \partial 1 \partial 2 \quad (3)$$

While evaluation of these integrals is straightforward, it is most efficiently carried out if integrals of a certain type are grouped together and all are evaluated in one batch. How many integrals form one batch depends on, among other things, the atom types in our system. In first instance most codes will continue to collect data until all integrals that can be grouped together are. The disadvantage of this is that it can lead to large buffers that overflow cache. In our modification we have effectively limited the maximum size of these buffers, so even though it may be possible to group together more integrals and save CPU operations, from a memory access perspective it is much better to split the batch into two or more smaller batches.

2.2. *Gaussian*[®] 98 Memory and Input/Output

As well as being able to consume large amounts of CPU time, *Gaussian*[®] 98 (and quantum chemistry codes in general) can require large quantities of memory and disk space. One class of algorithms that are particularly memory and disk hungry is second-order perturbation theory methods (commonly called MP2). In MP2 the dominant operation is an integral transformation, in which the four-index integrals defined in eqn.3 are transformed from one basis set representation to another. This process is shown below:

$$[ab|cd] = \sum_{ijkl}^N C_{ai}C_{bj}C_{ck}C_{dl}[ij|kl] \quad (4)$$

where the range of the $ijkl$ indices corresponds to the number of basis functions (N), but the range of the $abcd$ indices is less than this. For efficiency the above transformation is best performed as a series of four one-index transformations of the form:

$$[aj|kl] = \sum_i C_{ai}[ij|kl] \quad (5)$$

$$[ab|kl] = \sum_j C_{bj}[aj|kl] \quad (6)$$

The problem is that to form just one $[ab|cd]$ integral requires all $[ij|kl]$ integrals to be generated, and there are N^4 of these! While in practice this is not a problem as the $[ij|kl]$ integrals are generated and used immediately, this is not the case for the intermediate ‘‘partially transformed’’

integrals, e.g. the $[ab/kl]$ integrals. These must be either stored in memory or written to disk, and since there can be as many as N^3 of these the storage requirements can quickly become very large. For example for the benchmark systems given in Table 1 the number of basis functions ranges up to 882, and storing 882^3 double precision numbers requires in excess of 50GBytes.

In the past contemplating an MP2 calculation on a system with 882 basis functions would have been impossible because of the excessive CPU time required, but now such calculations are within our grasp. To this end part of our work on the Solaris version of *Gaussian*® 98 has been enabling the code to use large memory and create large disk files. Specifically in revision A.6 of the code the Solaris code was limited to files sizes of just 2Gbytes. This limit has been increased to 16GBytes in revision A.7, but will effectively be removed in the next release which will be based on using 8byte integers throughout the code.

2.3. Parallel *Gaussian*® 98

Gaussian® 98 exploits parallelism in three different ways, by using:

1. Coarse grain parallelism implemented via
 - `fork` or a similar utility to spawn multiple tasks that communicated via shared memory segments
 - Linda [ref] to create multiple tasks that communicate by data tuples
 - A combination of the above two methods
2. Loop level parallel directives
3. Parallellised basic linear algebra (BLAS) libraries

While all of the above are relevant to UltraSPARC platforms, most Sun users will not be concerned with the Linda parallel code as this is primarily intended for distributed memory environments. Furthermore, since Linda is a commercial product that must be purchased separately most users will not buy Linda unless they are forced to. Given this we will not discuss the Linda code further, and only consider the symmetric multiprocessor (SMP) code.

The coarse grain parallelism in *Gaussian*® 98 is associated with the generating and/or processing of integrals. Essentially all substantive integral generation tasks have been parallelised, and when a calculation enters one of these routines it automatically spawns the multiple tasks. The number of tasks created is determined by the user as an input parameter, with the default value being just one task. (We note that a different default number of tasks can be set as part of *Gaussian*® 98 installation). In the Solaris version of *Gaussian*® 98 the multiple tasks are generated using `fork`. As a consequence of this each task is an exact copy of the entire process at that point in time, and immediately has copies of all data items. The exception, however, is the main *Gaussian*® 98 scratch array. This array is dynamically allocated, with a size that can be set in the input deck. Normally the memory for this scratch array is allocated using `malloc`. If, however, multiple processes are being used this memory is allocated as a shared memory segment. (Note the size of this shared memory segment can be quite large, often above the default machine limit). Key arrays that must be unique to each of the forked processes are allocated to unique parts of this array. For instance each process uses a different part of the shared array to store a local copy of the Fock matrix (matrix F_{ij} in eqn.2). When, for example, each process has finished computing its local contribution to its own Fock matrix, all child processes terminate. The parent process then sums up the different Fock matrices to produce the final result.

A consequence of the communication model used by SMP parallel *Gaussian*® 98 is the need to keep multiple copies of various matrices and as a result the memory requirement increases as the number of processes increases. Indeed if the number of shared memory processors is progressively increased the code will eventually run out of memory if the size of the general scratch array is fixed. When this happens *Gaussian*® 98 will decrease the number of processors until it is able to run.

For many *Gaussian*® 98 calculations the coarse grain parallelism associated with the generation and/or processing of integrals parallelises over 90% of the total computational time. Unfortunately there are some calculations where this is not the case. For these calculations and for the parts of the code not affected by the coarse grain parallelism *Gaussian*® 98 uses loop level parallel directives and parallel numerical libraries. Unfortunately in the current A.7 release loop level parallelism is not activated for UltraSPARC platforms, in part this is because we encountered serious performance penalties associated with the generic use of the `-xexplicitpar` compiler directive. Work to fix this and to enhance the general level of parallelism in the code is underway.

The parallel performance of *Gaussian*® 98 revision A.7 on a 400MHz Sun Enterprise 3500 system is shown in Table 3 for the benchmarks given earlier. The speedup has been measured by comparing the elapsed time on multiple processors with the CPU time on a single processor. In all cases the memory allocated to the code was kept constant, which, as noted above resulted in some test jobs reducing the number of processors actually used. The results show that while some calculations are currently running fairly well in parallel, others are showing little or no parallel speedup. This situation will improve when the fine grain parallel directives are activated.

Table 3: Parallel Performance of *Gaussian*® 98 A.7 on a 400MHz Sun Enterprise 3500

u400	1CPU cpu	2CPU_ Elapsed	Speedup	4 CPU Elapsed	Speedup
test178	176	118	1.49	95 ^a	1.85
test397	59215	33317	1.78	16638	3.56
ealpb3	44466	21162	2.10	10752	4.14
falpb3	19953	10126	1.97	5549	3.60
falpm	79562	75052	1.06		
eisoqc	3347	4502	0.74		
ec20mp	2449	2662	0.92		
gc20mp	16867	18784	0.90		
gaphci	2826	1441	1.96	746	3.79
ebenca	1113	1096	1.02	1081 ^b	1.03

^a Reduces to 3 processes

^b Reduces to 3 processes

All times are seconds.

3. Performance of *Gaussian*® 98 on UltraSPARC

The challenge of benchmarking *Gaussian*® 98 is its memory and disk requirements. For best performance, big CPU caches, fast memory interconnect and big memory are the keys.

Unfortunately, not all of Sun's customers can afford that. That's where benchmarking enters to identify the best price/performance using the type of problem that the researcher typically solves.

3.1. Overall Performance

A subset of the benchmarks mentioned above has been run on a range of Sun servers from the Workgroup Server, Enterprise 450, to the Midrange Enterprise Servers, Enterprise 4500 and 6500 to the High-end Server, Enterprise 10000. Using these examples, it is easier to identify the type of system that is best.

Based on these sample tests, the mid-range Sun Enterprise Servers with 8MB E-cache are the best platform for running *Gaussian*[®] 98. The system clock speed boost on the E3500 and E4500 give those servers the edge. Tables 4 and 5 show the performance of Gaussian on different Sun systems and their relative performance. The 8MB E-caches give roughly a 10% speed up over the 4MB size. The faster memory system on the E450 gives it an advantage even with its smaller E-caches. The slower memory system on the E10000 causes a performance degradation of about 10% on a single job, however note that for very large throughput environments, where hundreds of jobs are competing for resources, the E10000 is faster than the mid-range servers overall.

Table 4: Performance on Various Sun Enterprise Servers

Benchmark	Run Type	Sun	Sun	Sun	Sun	Sun
		E3500	E450	E4500	E6500	E10000
		400/4MB	400/4MB	400/8MB	400/8MB	400/4MB
		G98 A.7	G98 A.7	G98 A.7	G98 A.7	G98 A.7
test178	SCF Energy	175	168	160	162	187
test397	DFT Force	57579	55115	52439	53047	65343
falpb3	DFT Hessian	19565	18843	17687	17884	21817
ec20mp	MP2 Energy	2464	2280	2264	2319	2816

All times are elapsed seconds. All runs were made using a single processor of a multi-processor system.

Table 5: Relative Performance on Various Sun Enterprise Servers

Benchmark	Run Type	Sun	Sun	Sun	Sun	Sun
		E3500	E450	E4500	E6500	E10000
		400/4MB	400/4MB	400/8MB	400/8MB	400/4MB
		G98 A.7	G98 A.7	G98 A.7	G98 A.7	G98 A.7
test178	SCF Energy	1.00	0.96	0.91	0.92	1.07
test397	DFT Force	1.00	0.96	0.91	0.92	1.13
falpb3	DFT Hessian	1.00	0.96	0.90	0.91	1.12
ec20mp	MP2 Energy	1.00	0.93	0.92	0.94	1.14

For ease of reference, Table 6 is a recap of the key characteristics of each of these Enterprise Servers.

Table 6: Configurations of the Sun Enterprise Servers

Computer	CPU Speed	System Clock	Number of CPUs	E-Cache Size	Physical Memory	Memory Interleave	Operating System
Sun E3500	400 MHz	100 MHz	8	4MB	8GB	Gigaplane™	Solaris 7
Sun E450	400 MHz	100 MHz	4	4MB	4GB	5-way UPA	Solaris 7
Sun E4500	400 MHz	100 MHz	14	8MB	8GB	Gigaplane™	Solaris 7
Sun E6500	400 MHz	80 MHz	20	8MB	26GB	Gigaplane™	Solaris 7
Sun E10000	400 MHz	100 MHz	64	4MB	64GB	Gigaplane-XB	Solaris 7

3.2. Effect of E-Cache Size

Gaussian® 98 is a cache unfriendly application. The bigger the cache and the faster the system clock the better for performance. The Table 7 shows the primary links for each of the four data sets whose times are shown in Table 4.

On the E3500 and the E4500 the system clock is the same, the external cache (E-cache) size is different. The 8MB E-cache is 10% faster!

Table 7: Effect of E-Cache Size

Benchmark	Run Type	Link	Sun	Sun	Speedup
			E3500 400/4MB G98 A.7	E4500 400/8MB G98 A.7	
test178	SCF Energy	502	151.5	137.7	1.1x
test397	DFT Force	401	209.5	177.8	1.2x
		502	48229.9	44126.5	1.1x
		703	9005.9	8015.0	1.1x
falpb3	DFT Hessian	502	1552.1	1427.4	1.1x
		1110	5391.0	5050.6	1.1x
		1002	6502.5	6083.8	1.1x
		703	6079.5	5089.7	1.2x
ec20mp	MP2 Energy	502	505.4	471.3	1.1x
		906	2360.8	2167.3	1.1x

All times are seconds. All runs were made using a single processor of a multi-processor system.

3.3. Effect of Memory Size on Parallel Performance

By default, *Gaussian*® 98 uses just a single processor and 48MB of memory. The researcher (or benchmarker) indicates the number of processors to use in one of two ways. *Gaussian*® 98 will not use all those processors, if it determines that there is not enough memory. A message prints in the output file telling the initial number of processors and, if that number was decreased, the number of processors it actually used.

Table 8 shows the impact of memory size on the number of processors actually used and the elapsed time. Also, notice that even if *Gaussian*® 98 is using all the requested processors, it may not impact the performance much; this is typical of this release. That will improve in future releases.

Table 8: Effect of Memory Size on Parallel Performance

Dataset	Memory	CPUs	Time (m:ss)
g98inp2 *	default	1	3:02
	30MB	1	3:02
	70MB	1	3:03
	default	2	1:47
	30MB	2	2:54 **
	70MB	2	1:50
	90MB	2	1:58
	default	4	1:46 **
	30MB	4	2:53 **
	70MB	4	1:24 **
	90MB	4	1:17
	110MB	4	1:15
	default	8	1:44 **
	30MB	8	2:53 **
	70MB	8	1:22 **

default = 48MB

* This dataset is not related to the benchmarks mentioned earlier.

** Used fewer than the number of requested CPUs because not enough memory was requested.

An interesting piece of information here would be how many CPUs were actually used for each run. That is difficult to extrapolate due to the fact that Gaussian reports when the number of CPUs is decreased, but does not report when that number is increased back to the requested number.

4. Conclusion

This joint project between ANU and Sun, although just beginning, has resulted in a better understanding of how the *Gaussian*® 98 algorithms work on Sun platforms and a 1.5x to 2.4x performance improvement on a single UltraSPARC processor. The next phase of this project implements 8 byte integers throughout the code and fine grain parallelism.

5. References

1. D.A. Dode, J.M. Cabral, R.A. Pfuetzner, A. Kuo, J. M. Gulbis, S.L. Cohen, B.T. Chait, and R. MacKinnon, "The Structure of the Potassium Channel: Molecular Basis of K⁺ Conduction and Selectivity", *Science*, 1998, v.230, pp.69-77.
2. J.J.P. Stewart, "Application of Localized Molecular Orbitals to the Solution of Semiempirical Self-Consistent Field Equations", *Int. J. Quant. Chem.*, 1996, v.58, p.133-146.
3. M. C. Strain, G. E. Scuseria and M. J. Frisch, "Achieving Linear Scaling for the Electronic Quantum Coulomb Problem," *Science* 271, 51 (1996).
4. M.J. Frisch, G.W. Trucks, H.B. Schlegel, G.E. Scuseria, M.A. Robb, J.R. Cheeseman, V.G. Zakrzewski, J.A. Montgomery, Jr., R.E. Stratmann, J.C. Burant, S. Dapprich, J.M. Millam, A.D. Daniels, K.N. Kudin, M.C. Strain, O. Farkas, J. Tomasi, V. Barone, M. Cossi, R. Cammi, B. Mennucci, C. Pomelli, C. Adamo, S. Clifford, J. Ochterski, G.A. Petersson, P.Y. Ayala, Q. Cui, K. Morokuma, D.K. Malick, A.D. Rabuck, K. Raghavachari, J.B. Foresman, J. Cioslowski, J.V. Ortiz, B.B. Stefanov, G. Liu, A. Liashenko, P. Piskorz, I. Komaromi, R. Gomperts, R.L. Martin, D.J. Fox, T. Keith, M.A. Al-Laham, C.Y. Peng, A. Nanayakkara, C. Gonzalez, M. Challacombe, P.M.W. Gill, B. Johnson, W. Chen, M.W. Wong, J.L. Andres, C. Gonzalez, M. Head-Gordon, E.S. Replogle and J.A. Pople, *Gaussian, Inc., Pittsburgh PA*, 1998.
5. E.K. Wilson, "Quantum Chemistry Software Uproar", *Chemical and Engineering News*, July 12th 1999, pp 27-30. OR H.F. Schaefer III, 5th World Congress of Theoretically Oriented Chemists, Imperial College, London 1999.