CMST 1(1) 7-18 (1996) DOI:10.12921/cmst.1996.01.01.07-18

The Role of Efficient Programming in Theoretical Chemistry and Physics Problems

Wojciech Cencek
Department of Chemistry, A. Mickiewicz University,
Grunwaldzka 6, 60-780 Poznań, Poland

Abstract

Various aspects of efficient programming of high-performance computer systems are discussed, using an example from modern electronic structure theory. It is shown that efficient programming is indispensable in today's theoretical studies by reducing drastically involved computer time. Timings from our quantum chemical program CORREL are given for several platforms ranging from Cray Y-MP EL to PC/486.

1 Introduction

1.1 Variational method in quantum mechanics

The quantum world is one of these fields where physics and chemistry meet particularly often. As it is commonly known, elementary particles and sufficiently small objects built out of them—such as atoms and molecules—cannot be described in terms of classical Newton mechanics, even when we neglect relativistic effects contained in Einstein's theory. Instead of Newton's laws we have to deal with the famous Schroedinger equation

$$\hat{H}\Psi = E\Psi, \tag{1}$$

where $\hat{H}$ is the so-called Hamilton operator modifying in some defined way the function Ψ following it. The construction of this operator for a given quantum system is usually a simple task. To solve the Schroedinger equation means to find both the "wavefunction", Ψ, and the total energy of the system, E. As follows from quantum mechanics, if we know Ψ we can easily calculate, apart from the energy, all static properties of the system. Unfortunately, from what the chemist may be interested in, namely atoms and molecules, only trivial one-electron cases such as the hydrogen atom can be directly solved. For larger systems the Schroedinger equation becomes so difficult that one is forced to look for some approximations. One commonly used, called the Born-Oppenheimer approximation, treats the movement of the electrons independently of this of the nuclei, basing upon the significant difference between masses of the two types of particles. In the first step, the **electronic** Schrödinger equation is solved, i.e. it is assumed that the nuclear masses in the Hamilton operator are infinite (the nuclei do not move) and the function Ψ depends only on the electronic coordinates. This is the subject of the **electronic structure theory** and we will restrict ourselves to this first step throughout this paper. The second step involves solving the equation for the nuclear motion. Having completed both

the tasks one can also subsequently compute the couplings between both the types of movements so that starting with the Born-Oppenheimer approximation does not exclude calculating the properties of the systems with high accuracy.

The electronic problem is also very complicated but can be numerically solved in some indirect way, for example **variationally.** Such methods rely on the following theorem (variational principle)

$$\varepsilon = \frac{\int \Psi \hat{H} \Psi \, dV}{\int \Psi \Psi \, dV} \geq E. \tag{2}$$

It states that taking *any* integrable function Ψ and calculating the quantity ε according to the above equation (where dV stands for the integration over all the electronic coordinates) one never gets a value lower than the true energy E. Consequently, among different trial functions the best one is that yielding the lowest value of ε. If these functions depend on some set of parameters, we treat ε as a function of these parameters and look for its minimum. It is particularly simple if Ψ is a linear combination

$$\Psi(\vec{r}_1, \ldots, \vec{r}_K) = \hat{P} \sum_{i=1}^{N} c_i \phi_i. \tag{3}$$

In the above equation, $\vec{r}_i$ denotes the set of all the coordinates of the ith electron and $\hat{P}$ is some operator ensuring that the total function has appropriate symmetry properties. A standard way in the electronic structure theory is to express the ϕ_i's as products of K one-electron factors, which is known as the method of configuration interaction (CI). It turns out, however, that the CI expansions are extremely slow convergent, i.e. the value of ε in Eq. (2) is far above E (at least dozens of cm $^{-1}$) unless an astronomic (computationally not achievable) number of expansion terms, N, is used. Higher accuracy can be obtained when the ϕ_i's depend not only on the individual electron coordinates but also on interelectronic distances. These so-called **explicitly correlated** functions describe better electron-electron Coulomb repulsions. Examples are the Kołos-Wolniewicz function |1] for the hydrogen molecule (one of the greatest achievements of the Polish chemistry) and the Gaussian-type function introduced in the 60's by Boys [2] and Singer [3|:

$$\phi_i = \exp\left(-\sum_{k=1}^{K} \alpha_{ik} |\vec{r}_k - \vec{A}_{ik}|^2 - \sum_{k>l=1}^{K} \beta_{ikl} |\vec{r}_k - \vec{r}_l|^2 \right) \tag{4}$$

The first sum contains electron coordinates expressed as squared distances between the kthelectron and the given point $\vec{A}_{ik}$. The second sum mixes the electron coordinates introducing explicit correlation of their movements. Because of the symmetry requirements, in atoms all the centers $\vec{A}_{ik}$ coincide usually with the position of the nucleus and need not to be determined, and in linear molecules they lie on the molecular axis and have one component to be optimized. Apart of these, with each ϕ_i term the following variational parameters are connected: one c_i (linear), K α_{ik}'s (nonlinear) and $\frac{1}{2}K(K-l)$ β_{ikl}'s (nonlinear). This type of wavefunction has been tried for thirty years, generally yielding significantly worse results than other explicitly correlated functions (e.g. the Kołos-Wolniewicz expansion). An opinion that it is inferior—when very high accuracy is desired—has been commonly accepted. A few years ago a large project was started in our group, which aimed at weakening this conviction and investigating what level of accuracy can be reached when large expansions with very carefully optimized nonlinear

parameters are used in conjunction with high-performance and efficiently programmed computers. It should be noted that almost all of today's quantum chemical computations are restricted to the linear optimization because optimizing nonlinear variational parameters is usually extremely expensive. As a part of the project we developed a computer program named CORREL which allows to optimize all the parameters in the wavefunction of the type (4) for small atoms and molecules. The largest expansion used so far by us has been a 2172-term wavefunction for the lithium atom ($N = 2172$, $K = 3$). As follows from above, it contains 2172 linear and 13032 nonlinear parameters, all of them to be optimized. This already sheds some light on the difficulties arising from such extensive calculations and explains the need of efficient programming.

1.2 Optimization of the linear parameters

In case we would like to optimize only the linear parameters c_i in (3), the problem is not theoretically difficult and has a general solution. Putting (3) into the expression for ε (2) and assuming $\partial\varepsilon/\partial c_i = 0$ leads to

$$\mathbf{H}\,\mathbf{c} = \varepsilon\,\mathbf{S}\,\mathbf{c} \tag{5}$$

where H and S are $N \times N$ matrices defined by

$$h_{ij} = \int \phi_i \hat{H} \phi_j \, dV \tag{6}$$

$$s_{ij} = \int \phi_i \phi_j \, dV \tag{7}$$

and **c** is an $N \times 1$ vector of the parameters c_t (see for example [5]).

Equation of the type (5) is well-known in linear algebra as the **general symmetric eigenvalue problem.** Its solution presents no formal difficulties but becomes time-consuming for large matrices since its cost goes as N^3.

It should be also noted that Eq. (5) has N solutions defined by the pairs $\{(\varepsilon, \mathbf{c})_l,\ l = 1, N\}$, each of them fulfilling the variational principle (2) for the /th quantum state of the system (l=1 for the ground state and $l \geq 2$ for excited states).

1.3 Optimization of both the linear and the nonlinear parameters

Unfortunately, it is not possible to obtain a compact equation for the best nonlinear parameters, analogous to (5). Instead we are forced to use some kind of nonlinear optimization. There are many such techniques but most of them consist in changing iteratively the values of the parameters according to some predefined strategy, testing the function (in our case—energy) value after each step, and modifying the strategy in order to move in the direction where the energy diminishes faster. The process stops when no modification of the parameters leads to further energy improvement. In the CORREL program we have implemented the so-called conjugate directions method introduced thirty years ago by Powell [4] and still considered as one of the most efficient despite its simplicity.

The whole algorithm of optimizing all the parameters in the wavefunction of the type (3) can be summarized as follows

1. Choose starting values for the nonlinear parameters.

2. Compute **H** and **S** matrices.

3. Solve eigenvalue problem $\mathbf{Hc} = \varepsilon \mathbf{Sc}$.

4. Stop condition fulfilled?
   NO: change the values of the nonlinear parameters; go to 2.
   YES: stop.

As we can see, each (2,3,4)-loop iteration in the optimization of the nonlinear parameters involves the full linear optimization (step 3). In practice a number of these iterations which are necessary to reach convergence turns out to be of the order of 100 times the number of the nonlinear parameters. Turning back to our 2172-term example lithium function, it means that following tasks have to be completed *over one million times:*

- computation of more than 2 millions matrix elements h_{ij} and s_{ij}, each of them involving calls to several complicated procedures,
- solving generalized eigenvalue problem of the order N=2172.

It is worth to say that doing the latter just *once* was itself a serious computational challenge only some 10 years ago. It becomes clear that without clever and efficiently programmed algorithms our method would be hopelessly time-consuming.

2 Details of the algorithms

2.1 Inverse iteration. Theory

Solving the eigenvalue problem (5) is crucial for the efficiency of our program. Even if, for some special cases, computing **H** and S matrices might be the slowest step, its time grows only as N^2. For high-accuracy quantum chemical calculations, in which we are ultimately interested, *N* has to be large and N^3-dependence of (5) will sooner or later provide a bottleneck.

We have used a method called inverse iteration [6], known as fast and numerically stable. The vector c, being a solution of Eq. (5), is found as the following limit

$$\mathbf{c} = \lim_{k\to\infty} \frac{\mathbf{c}_k}{|\mathbf{c}_k|}, \tag{8}$$

where the sequence $\{\mathbf{c}_k\}$ is defined by the recurrence relation

$$(\mathbf{H} - \varepsilon_0 \mathbf{S})\, \mathbf{c}_{k+1} = \mathbf{S}\mathbf{c}_k \tag{9}$$

and c_0 is an arbitrary starting vector. It can be proven [6] that $\{\mathbf{c}_k\}$ converges to an eigenvector corresponding to this eigenvalue which is closest to the constant ε_0. Therefore, the eigenvalue (energy) has to be known approximately at the beginning. The exact value can be computed from the converged eigenvector:

$$\varepsilon = \frac{\mathbf{c}^T \mathbf{H}\mathbf{c}}{\mathbf{c}^T \mathbf{S}\mathbf{c}}. \tag{10}$$

In practice, for a reasonably chosen ε_0, only a few iterations $(\sim 3 \div 5)$ are needed to obtain 15 significant digits of ε.

2.2 Inverse iteration. Computational scheme

That the computational effort involved in the inverse iteration procedure is proportional to N^3 comes from the necessity to solve (several times) the linear equations system (9). Such systems, very common in numerical linear algebra, can be generally written as

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{11}$$

where the matrix **A** and the vector **b** are known. In our case, we have to solve such a system once in each iteration, i.e. for each value of of k in (9). It can be also noted that $\mathbf{b}=\mathbf{S}\mathbf{c}_k$ changes from one iteration to another and has to be recalculated each time, whereas $\mathbf{A}=\mathbf{H}-\varepsilon_0\mathbf{S}$ remains constant. This observation is very useful due to the following well-known facts. Firstly, if the matrix in the linear equations system is either lower triangular L (has zeroes everywhere above the diagonal) or upper triangular U (has zeroes everywhere below the diagonal)

$$\mathbf{L}\mathbf{x} = \mathbf{b} \qquad \text{or} \qquad \mathbf{U}\mathbf{x} = \mathbf{b}, \tag{12}$$

then solving such a system requires only ~ N^2 operations instead of ~ N^3. Secondly, every matrix **A** can be decomposed with ~ N^3 operations into a product of a lower triangular matrix **L** and an upper triangular matrix **U**

$$\mathbf{A} = \mathbf{L}\mathbf{U} \tag{13}$$

what allows to express $\mathbf{A}\mathbf{x}=\mathbf{b}$ equivalently as

$$\mathbf{L}\mathbf{y} = \mathbf{b} \tag{14}$$

$$\mathbf{U}\mathbf{x} = \mathbf{y}. \tag{15}$$

Now it becomes clear that we can decompose $\mathbf{A} = \mathrm{H} - \varepsilon_0\mathbf{S}$ at the beginning and in subsequent iterations solve much less expensive systems (14) and (15) instead of (11). The slowest step—matrix decomposition—remains proportional to N^3, but it has to be done just once. The final scheme of the inverse iteration procedure reads as follows.

1. Calculate $\mathbf{A} = \mathbf{H} - \varepsilon_0\mathbf{S}$.
2. Find the decomposition: $\mathbf{A}=\mathbf{L}\mathbf{U}$.
3. $k \leftarrow 0$.
4. Choose the starting vector c_k.
5. Calculate $\mathbf{S}\mathbf{c}_k$.
6. Solve $\mathbf{L}\mathbf{y}=\mathbf{S}\mathbf{c}_k$.
7. Solve $\mathbf{U}\mathbf{c}_{k+1}=\mathbf{y}$.
8. $\mathbf{c}_{k+1} \leftarrow \mathbf{c}_{k+1}\left(\mathbf{c}_{k+1}^T\mathbf{S}\mathbf{c}_{k+1}\right)^{-1}$
9. Calculate the eigenvalue $\varepsilon = \mathbf{c}_{k+1}^T\mathbf{H}\mathbf{c}_{k+1}$.
10. Stop condition fulfilled?
    (ε changed negligibly between last two iterations?)
    NO: $k \leftarrow k+1$; go to 5.
    YES: stop.

2.3 Updating algorithms: A clue to the efficiency

Although we chose a very good procedure to solve the general eigenvalue problem (5), it still requires ~ N^3 operations and has to be invoked thousands or millions times to get the optimized nonlinear parameters (see Sec. 1.3). Fortunately, the only N^3-dependent step (matrix decomposition A = L U) has some very convenient property which we will now employ to reduce dramatically the optimization cost. Namely, elements of the kth row of L (and the *kth* column of U) are functions of elements of the first k rows and columns of A but do not depend of whatever is in rows (columns) $k + 1$, $k + 2$, ..., N of A. Let us now figure that we have to decompose some sequence of matrices A_1 A_2, ..., A_n which differ only in the last row and column. Since rows 1, 2, ..., up to k - 1 of L and columns 1, 2, ..., up to k - 1 of U do not depend on what is in the last row and column of A, they will be the same along the whole sequence. As a consequence, only the first matrix requires the full decomposition and ~ N^3 operations. For the other matrices, only one single row of L and one single column of U remain to be calculated, which is of course much faster and depends on N as N^2. This is an example of what is called **updating** in numerical linear algebra, namely using information accumulated during the solution of a problem to solve faster the same problem slightly modified, or *perturbed.*

How can we ensure that the matrix $\mathbf{A}=\mathbf{H}-\varepsilon_0\mathbf{S}$ does not change during the optimization process except in the last row and column? First of all, only parameters belonging to a single term of expansion (3) may change simultaneously in one iteration (one pass through steps 2, 3, 4 in Sec. 1.3 is meant here as the iteration). In other words, we optimize one term ϕ_i while keeping the rest fixed. Then, only term $i + 1$ is optimized and so on. But the matrix should have changed at the end, not somewhere in the middle, shouldn't it. Here the solution is very simple. Before we start to optimize the ith term, we formally renumber the terms—this can always be done without changing the solutions of (5)—so that the ith and the last trade their places. Optimization of a single term typically requires a few dozens iterations. However, only the first iteration involves the standard, N^3-dependent inverse iteration procedure. All the remaining ones can be done considerably faster thanks to the updating.

Optimizing only a single function ϕ_i has one more advantage: We do not need to recompute the whole H and S matrices (see step 2 in Sec. 1.3), since only one row and column change from one iteration to another. On the other hand, such an optimization is less flexible: The parameters in the ith term, no matter how carefully they were obtained, are optimal only with respect to the current fixed values of the others. They are not optimal as soon as the next term is released and modified. Therefore, going once from i-1 to i=N is not sufficient. In our program dozens or hundreds of such cycles are repeated until a satisfactory convergence is reached.

3 Practical implementation

3.1 General remarks

The inverse iteration algorithm described in Sec. 2.2, with updating capabilities added, has been coded in Fortran 77 as a procedure named SOLVE, a part of our CORREL program. Running CORREL on different platforms we have always taken care of tuning this crucial procedure to the particular hardware and software environment. It was soon realized that SOLVE can be a good test of computing performance for vector problems.

On the contrary, we have used a module generating matrix elements (6) and (7) to measure scalar performance, which will be discussed later.

For each platform we have prepared, whenever possible, three different versions of the SOLVE procedure:

1. containing only standard Fortran 77 statements and intrinsic function calls, compiled with maximum scalar optimization but without using vector or pipelining capabilities
2. containing only standard Fortran 77 statements and intrinsic function calls, compiled with maximum scalar and vector (or pipeline) optimization
3. containing calls to hardware-optimized library routines

Parallelization of SOLVE on multiprocessor machines requires special programming techniques and is beyond the scope of this paper. All the tests described here were run on a single processor.

3.2 Tested systems and compiler options

These are the systems on which we ran SOLVE, the compilers used and the compiler options, chosen after a number of tests. The first options line gives the invocation of the compiler for version 1 of SOLVE, the second one for 2 and 3.

- Cray Y-MP EL 33 MHz
  Compiler: Cray CF77 Version 6.0 (6.49)
  cf77 -dp -Oscalar3 -OvectorO
  cf77 -dp -Oscalar3 -Ovector3
  Remarks: All inner loops have been vectorized without need to change the source code

- Silicon Graphics Power Challenge L (processor MIPS R8000 75 MHz)
  Compiler: MlPSpro F77 6.0
  f77 -64 -mips4 -03 -sopt -SWP:=OFF
  f77 -64 -mips4 -03 -sopt -SWP:=ON
  Remarks: Linking version 3 requires -lblas option

- Hewlett-Packard 715 50 MHz
  Compiler: HP-UX Fortran/9000 09.16
  f77 +04 +OP4 +Onopipeline
  f77 +04 +OP4
  Remarks: Version 3 has been not prepared (BLAS and matrix libraries unavailable)

- Microway Number Smasher XR-860 (processor Intel i860 40 MHz)
  Compiler: Microway NDP Fortran-860 4. Id for DOS
  mf860n -on
  mf860n -on -vast
  Remarks: Vectorizing preprocessor Pacific Sierra Research VAST-2 was used for version 2. All inner loops have been vectorized without need to change the source code

- PC Pentium (processor Intel Pentium 90 MHz)
  Compiler: Microway NDP Fortran-486 4.2.0 for DOS
  mf486 -n2 -n3 -on -486 -4861ib
  Remarks: An older version of NDP compiler without Pentium-specific optimizations was used. SOLVE versions with pipelining or optimized libraries unavailable.

3.3 Hardware-specific libraries

In three cases (Cray, Power Challenge and Number Smasher) we have used libraries containing routines callable from Fortran. Most of linear algebra routines available on different systems belong to the following classes:

1. BLASl (Basic Linear Algebra Subroutines Level 1): vector-vector operations
2. BLAS2: matrix-vector operations
3. BLAS3: matrix-matrix operations
4. LINPACK: high-level standard procedures such as matrix decomposition or solving linear systems of equations; invoke BLASl and BLAS2 routines
5. LAPACK: a modern package of routines developed to replace LINPACK; invoke mostly BLAS3 routines

Generally speaking, a user program should call machine-specific routines of the highest possible level to get best efficiency (one can imagine a whole user program written directly in the machine language as the limiting case). Therefore, when possible BLAS3 should be preferred to BLAS2 or BLASl and LAPACK is more efficient than LINPACK.

The list below contains library routines used in Cray version 3 of SOLVE

. SPOTRF (LAPACK)
Decomposes a symmetric positive definite matrix into triangular matrices with the Cholesky method.

- SPOTRS (LAPACK)
  Solves a system of linear equations using triangular matrices generated by SPOTRF.
- SSYMV (BLAS2)
  Multiplies a symmetric matrix by a vector.
- STRMV (BLAS2)
  Multiplies a triangular matrix by a vector.
- SDOT (BLASl)
  Computes a scalar product of two vectors.
- SSCAL (BLASl)
  Multiplies a vector by a scalar.
- SCOPY (BLASl)
  Copies a vector into another vector.

Analogous routines are available for Number Smasher. We have used CLASSPACK package from Kuck & Associates, probably the best libraries for Intel i860 processor.

At the time of our tests, we had access only to BLASl and BLAS2 routines for the Power Challenge machine. They are linked to a user program with -lblas option.

No linear algebra libraries for Hewlett-Packard and Pentium machines were available to us during the tests.

4 Results of the tests

4.1 Timings from the vector test

Tables 1 and 2 contain CPU times required to find the lowest root of the general symmetric eigenvalue problem with different versions of the SOLVE procedure on different machines. Times in parentheses concern solving the same problem using updating.

Table 1: SOLVE timings for N=125 and N=250. All times in seconds.

N=125	scalar mode	vector mode	with libraries
Cray Y-MP EL	0.246 (0.066)	0.118 (0.019)	0.020 (0.007)
Power Challenge	0.029 (0.008)	0.013 (0.003)	0.013 (0.003)
NS 860 40 MHz	0.153 (0.040)	0.134 (0.028)	0.059 (0.017)
HP 715 50 MHz	0.106 (0.034)	0.066 (0.027)	
Pentium 90 MHz	0.130 (0.032)		

N=250	scalar mode	vector mode	with libraries
Cray Y-MP EL	1.483 (0.240)	0.504 (0.045)	0.089 (0.019)
Power Challenge	0.187 (0.032)	0.070 (0.012)	0.070 (0.012)
NS 860 40 MHz	0.955 (0.154)	0.691 (0.086)	0.285 (0.050)
HP 715 50 MHz	0.922 (0.128)	0.674 (0.105)	
Pentium 90 MHz	0.827 (0.135)		

As it could be expected, the vectorized (pipelined) versions are always faster than scalar ones but inferior to those calling library routines. Cray, a typically vector machine, benefits most from non-scalar optimization techniques: The slowest in scalar mode becomes the fastest with specialized libraries and large matrices (N=1000), but—surprisingly enough—loses with Power Challenge for smaller values of N. As can be seen, the speed-up resulting from the updating technique grows with the matrix size and for N=1000 amounts to more than one order of magnitude.

4.2 Timings from the scalar test

The scalar test has consisted in computing the full lower triangles of the symmetric matrices H and S, defined by (6) and (7), for a 300-term expansion of the H_2 molecule wavefunction, i.e. 45150 matrix elements h_{ij} and s_{ij}. The relevant module of the CORREL program contains long sequences of arithmetic floating point operations, calls

Table 2: SOLVE timings for N=500 and N=1000. All times in seconds.

N=500	scalar mode	vector mode	with libraries
Cray Y-MP EL	10.00 (0.912)	2.320 (0.119)	0.513 (0.059)
Power Challenge	1.405 (0.162)	0.502 (0.082)	0.445 (0.048)
NS 860 40 MHz	6.620 (0.618)	3.972 (0.291)	1.730 (0.165)
HP 715 50 MHz	6.707 (0.498)	4.820 (0.407)	
Pentium 90 MHz	5.935 (0.538)		

N=1000	scalar mode	vector mode	with libraries
Cray Y-MP EL	73.36 (3.582)	11.89 (0.359)	3.444 (0.205)
Power Challenge	12.50 (0.838)	5.970 (0.577)	5.600 (0.423)
NS 860 40 MHz	50.59 (2.608)	24.71 (1.077)	12.08 (0.605)
HP 715 50 MHz	50.16 (1.995)	35.05 (1.635)	
Pentium 90 MHz	45.86 (2.398)		

to the intrinsic exponential function and no loops. The best times from each machine (additionally, a PC/486 system has been tested) are summarized in Table 3. The last place of Cray Y-MP EL may be perhaps somewhat shocking for a non-specialist but it confirms once more that the power of Cray comes from its vector architecture.

Table 3: Timings from the scalar test. All times in seconds.

HP 715 50 MHz	4.3
Power Challenge	4.8
Pentium 90 MHz	7.5
486 DX/2 66 MHz	16.4
NS 860 40 MHz	16.6
Cray Y-MP EL	21.7

4.3 Summary of the tests

We have just seen that the relative performance of various computer systems depends significantly on the nature of the problem (vector or scalar). This effect can be expressed semi-quantitatively by introducing the "vector character" of a system, defined by

$$\eta = \frac{(t_s/t_v)}{(t_s/t_v)_{\min}}, \tag{16}$$

where t_s is the best time of solving the scalar problem on the particular system and t_v is the best time to solve the eigenvalue problem for N=1000 using updating on the same system. The values of t_s/t_v are normalized so that the lowest possible value of η is 1. The larger is η, the faster is the system when crunching vector problems compared

to its speed doing only scalars. It must be stressed that we have defined this quantity only for the purpose of our present discussion and it is not strict because someone else taking other vector and scalar tests would obtain a little different values. Nevertheless, the conclusions would be the same. Again, as we can see in Table 4, Cray turns out to be "the most vector" of all the tested machines. What does the value η=40.2 mean in practice? Imagine that someone asks the following question: "How many times is Cray Y-MP EL faster than Hewlett-Packard 715/50?" His friends take their programs and make the tests. Depending on what programs they use, their answers can differ by a factor of forty!

Table 4: Vector character η of different computer systems.

system	η
HP 715 50 MHz	1.0
Pentium 90 MHz	1.2
Power Challenge	4.3
NS 860 40 MHz	10.4
Cray Y-MP EL	40.2

Strictly speaking, the calculated values of η for HP and Pentium systems are too low because we didn't have the relevant libraries which would speed-up the vector computations. At least in the Pentium case it would hardly change our conclusions qualitatively because this processor does not seem to benefit from the pipelining so much as typical RISC chips.

Taking into account the collected data we can formulate the optimal application areas of these systems:

- HP 715/50: an ideal workstation for extensive scalar computations,
- Pentium: a low-cost system for scalar computations; very good performance/price ratio,
- Power Challenge: a universal tool with very well balanced vector and scalar power; ideal for mixed problems,
- Number Smasher: a low-cost system for vector computations; offers one-third of the vector power of Cray Y-MP EL for $4000!
- Cray Y-MP EL: ideal for large-scale vector problems; **used for scalar applications only by barbarians;** it can be done faster on a PC!

5 Conclusions

We spent indeed a lot of time making CORREL more and more efficient, and finally let us say a few words about profits—from the physicist and chemist's point of view, not the programmer's. CORREL has been employed so far to investigate the following systems:

- 2-electron: ground [7] and several excited states of H_2, ground states of HeH^+ [8] and H_3^+
- 3-electron: H_3 [9], He_2^+ [10], Li
- 4-electron: LiH [9], Be, He_2

It is sufficient to say that in each case we obtained lower (i.e. more accurate) energies than previously reported variational results. The result for the ground state of H_2 is better than those obtained with the Kołos-Wolniewicz function and represents in fact the highest level of accuracy ever reached in molecular quantum computations (except for trivial one-electron cases). These facts change dramatically the reputation of the Gaussian-type explicitly correlated wavefunctions (4) in quantum chemistry (see Sec. 1.3).

In conclusion let us summarize what can be learnt from our experience with developing the CORREL program.

- Efficient programming allows to reduce drastically time needed to get the results, often even making the given approach applicable.
- Importance of various elements of the efficient programming can be ranked as follows

  optimal algorithm and its coding > system libraries > vectorization

- There is no such notion as speed of a computer system. Instead, we should talk about the speed of the system applied to the particular problem.

Acknowledgments

This work was supported by the KBN grant 8 T11F 010 08p01. The author wishes also to thank Poznań Supercomputing and Networking Center for the computer time.

References

[1] W. Kołos, L. Wolniewicz, *J. Chem. Phys.* **43,** 2429 (1965).

[2] S. F. Boys, *Proc. Roy. Soc.* **A258,** 402 (1960).

[3] K. Singer, *Proc. Roy. Soc.* A258, 412 (1960).

[4] M. J. D. Powell, *Comput. J.* 7, 155 (1964).

¡5) W. Kołos, *Chemia kwantowa,* PWN Warszawa, 1986 (in Polish).

[6] A. Kiełbasiński, H. Schwetlick, *Numeryczna algebra liniowa. Wprowadzenie do obliczeń zautomatyzowanych,* Wydawnictwa Naukowo-Techniczne, Warszawa 1992 (in Polish).

[7] J. Rychlewski, W. Cencek, J. Komasa, *Chem. Phys. Lett.* 229, 657 (1994).

[8] **J.** Rychlewski, *Int. J. Quant. Chem.* **49,** 477 (1994).

[9] W. Cencek, J. Rychlewski, *J. Chem. Phys.* **98,** 1252 (1993).

[10] W. Cencek, J. Rychlewski, *J. Chem. Phys.* **102,** 2533 (1995).