NMR Chemical Shift Calculations on Parallel Platforms

Bernd Pfrommer
Department of Physics
University of California at Berkeley
501 Birge Hall
Tel. 510 642-2635

June 6, 1997

Abstract

We present a parallel implementation of our new method to compute the nuclear magnetic resonance (NMR) chemical shift in condensed matter. Our theory uses a quantum-mechanical framework to compute the magnetic field induced by a uniform external magnetic field. While it has been previously possible to compute the chemical shift of finite systems such as molecules only, our method allows us to treat infinitely extended systems such as crystals, liquids and amorphous solids. By solving a system of linear equations rather than computing the full spectrum of a matrix, we are able to handle hundred atoms and more. From the beginning, the algorithm and implementation were designed to be parallel. We devise a blocked conjugate-gradient solver for better cache utilization. The three-dimensional Fast Fourier Transform (FFT) involved in Matrix-vector products is the most difficult to parallelize. We describe the data layout, and show that our implementation using the MPI message passing library and Fortran 90 gives good efficiency and portability. Performance numbers for the Cray T3E, IBM SP2, and SGI PowerChallenge are presented and discussed.

1 Chemical Shift Calculations – Outline of the Algorithm

It is the intend of this section to introduce the subject to a point that the reader can understand the origin of the algorithms. We will be overly simplifying, schematic and sometimes strictly incorrect in the presentation to avoid cluttering details.

The induced magnetic $\mathbf{B}_{in}$ is related to the external, applied magnetic field $\mathbf{B}_{ext}$ by

$$\mathbf{B}_{in}(\mathbf{r}) = 4\pi \int \chi(\mathbf{r}, \mathbf{r}')\mathbf{B}_{ext} dr' \ , \tag{1}$$

where χ is the magnetic susceptibility tensor, which depends on the position of the induced field $\mathbf{r}$, and the position $\mathbf{r}'$ of the external field. Its spatial variation means that different nuclei will feel a different induced magnetic field, and hence exhibit a characterstic "chemical shift" when probed with NMR techniques. Thus, NMR is widely used to identify substances, but is also useful for structural analysis.

Our goal is to compute the tensor χ, but we do so in Fourier space, since we will be dealing with infinite periodic systems such as crystals. If they are not periodic (such as molecules and amorphous systems), we can always make them periodic by a replication at sufficiently large distances.

Therefore, we work mostly in Fourier-space. For ease of exposition, we will stay in $\mathbf{r}$-space here. By definition, χ is computed as the second derivative of the (quantum-mechanical) energy E as a function of the magnetic field:

$$\chi(\mathbf{r}, \mathbf{r}') = -\frac{\delta^2 E[\mathbf{B}]}{\delta \mathbf{B}(\mathbf{r}) \delta \mathbf{B}(\mathbf{r}')} \ . \tag{2}$$

The energy E depends on the magnetic field via the equations of quantum mechanics, where we compute the second derivative via second order perturbation theory:

$$\chi \propto \sum_v \sum_c \frac{\langle \phi_v | H^{(1)} | \phi_c \rangle \langle \phi_c | H^{(1)} | \phi_v \rangle}{\epsilon_v - \epsilon_c} \ . \tag{3}$$

Here, the $|\phi_v\rangle, v = 1 \ldots m$ are the valence band wave functions, i.e. those quantum-mechanical states occupied by electrons, and $\phi_c\rangle, c = m+1 \ldots N$ are the conduction band wave functions, which are unoccupied. The effect of the magnetic field is contained in the "perturbation" operator $H^{(1)}$. The wave functions are solutions to a Schrödinger-type eigenvalue equation for the hermitian Hamiltonian operator H:

$$H|\phi_i\rangle = \epsilon_i |\phi_i\rangle \ , \qquad\qquad i = 1 \ldots N \ , \tag{4}$$

which has many solutions, ascendingly ordered by their (energy) eigenvalues ϵ_i. When expanding the wave functions $|\phi_i\rangle$ in a basis set of size N, (4) turns into a large matrix equation for which the $|\phi_i\rangle$ are eigenvectors. We will henceforth call the wave functions "vectors", because they are represented by a vector of expansion coefficients with respect to a (Fourier) basis set. The size N of the matrix can easily reach 50000 or more. But do we really need all its solutions $|\phi_i\rangle, i = 1, \ldots N$, or is it enough to just compute the much fewer valence wave functions $|\phi_v\rangle, v = 1, \ldots m$? Typically, m is at most a few hundred, and thus much smaller than N. A few lowest eigenvalues and eigenvectors can be efficiently computed by means of an iterative algorithm. Computing *all* the eigenvectors is very demanding both computationally and in terms of memory (just storing all the eigenvectors would require 40 GB of memory). The problem is solved by realizing that Eq. (3) can be rewritten as:

$$\chi \propto \sum_v \langle \phi_v | H^{(1)} | \psi_v \rangle \ , \tag{5}$$

with the definition:

$$|\psi_v\rangle = \sum_c \frac{|\phi_c\rangle \langle \phi_c | H^{(1)} | \phi_v \rangle}{\epsilon_v - \epsilon_c} \ . \tag{6}$$

Now the problem is to compute $|\psi_v\rangle$ in Eq. (6), which still requires the knowledge of all $|\phi_i\rangle$. But there is an alternate route to computing the $|\psi_v\rangle$'s. In fact, they are solutions to the equation:

$$(H - \epsilon_i I)|\psi_i\rangle = -\left(I - \sum_v |\phi_v\rangle\langle\phi_v|\right) H^{(1)} |\phi_i\rangle \qquad v = 1, \ldots N. \tag{7}$$

Equation (7) is a large system of linear equations of the form $A\mathbf{x} = \mathbf{b}$, and has to be solved for each of the $|\psi_i\rangle, i = 1, ..m$. Since there are efficient iterative schemes for solving linear equations, this is about as expensive as computing the valence wave functions $|\phi_v >, v = 1, \ldots m$. However, we now avoid the prohibitive cost of computing the rest of the spectrum $|\phi_c >, c = m+1, \ldots N$. This trick enables us to handle much larger problems.

We summarize the section by presenting the two key algorithms which we use to compute the NMR chemical shift.

1. A conjugate-gradient based iterative algorithm[2] to find the m lowest eigenvectors $|\phi_v\rangle$ and eigenvalues ϵ_v of the large $N \times N$ matrix H in Eq. (4). Typically m is a few hundred, but N is several tens of thousands.

2. A conjugate-gradient solver to find the m solutions $|\psi_v\rangle$ to the m large $N \times N$ linear systems of Eq.(7).

As far as a performance analysis is concerned, these are the most important routines. However, they constitute only a small part inside a quantum-chemistry type of code which the author has parallelized, enlarged and maintained for the last three years. It has meanwhile grown from about 22,000 lines to 33,000 lines (including comments), and runs completely in parallel on several different platforms.

2 Computational Kernels

Fortunately, there are only two performance critical operations that the linear system solver and the iterative eigen solver require:

- The multiplication of an arbitrary vector $|\varphi\rangle$ with the matrix H:

$$|\varphi'\rangle = H|\varphi\rangle \ . \tag{8}$$

  Due to the special representation of H, this operation involves matrix-matrix multiplies and a FFT, as explained below.

- Operations like dot products between two arbitrary vectors $|\varphi\rangle$:

$$\langle\varphi_i|\varphi_j\rangle = S_{ij} \ . \tag{9}$$

  These operations are only required for the iterative eigensolver, for instance to achieve orthogonality between the different eigenvectors. By computing (9) for all i and j simultaneously, this turns into matrix-matrix multiplies, so we can implement (9) efficiently as level 3 BLAS calls.

The matrix-vector product $H|\varphi\rangle$ is the more difficult part of the two kernels. We can cut the computational cost and avoid the storage of H by exploiting its special structure. As an operator, it can be written as:

$$\hat{H} = -\frac{1}{2}\nabla^2 + \sum_{k=1}^{n_{nloc}} (-1)^{\alpha_k} |\Phi_k\rangle\langle\Phi_k| + V_{loc}(\mathbf{r}) \tag{10}$$

Since we keep the vectors in Fourier space, the $-\frac{1}{2}\nabla^2$ term is a diagonal matrix, and easy to apply. The second term ("nonlocal potential") is a sum of $n_{nloc} \approx m$ rank one matrices, which is also simply applied in Fourier space as a matrix-matrix multiplication. The local potential term $\hat{V}_{local}$ in (10) is diagonal in a realspace basis. Therefore, it is applied to a wave function $|\varphi\rangle$ by first inverse Fourier transforming $\varphi(\vec{G}) \rightarrow \varphi(\vec{r})$ (operation count $\mathcal{O}(N \log N)$), then multiplying with a diagonal operator in realspace (operation count $\mathcal{O}(N)$), and finally Fourier transforming back to the plane wave basis. Since $n_{nloc} \approx m$, and m and N are proportional to the number of atoms N_{atom}, the second term applied to a set of wave functions requires $\mathcal{O}(N_{atom}^3)$ operations. The local

term only scales like $\mathcal{O}(N_{atom}^2 \log N_{atom})$, but has a large prefactor in front, such that it dominates the computational cost for systems with less than 50 atoms.

Optimizing the kernel separately from the algorithm can lead to poor performance. We realized that the Fourier transform involved in $H|\varphi\rangle$ has a large startup cost for bringing the real space work arrays and the local potential operator $V_{loc}(\mathbf{r})$ into cache. We gained a factor of 4 improvement by writing a blocked version of the conjugate gradient linear solver that solves all m equations in (7) simultaneously, and therefore applies $H|\varphi_i\rangle, i = 1, ..m$ to the full set of wave functions. This results in better cache utilization and avoids the low memory bandwidth of RISC-based architectures. Such an algorithmic change was *not* necessary for the conjugate gradient eigen solver, since all operations there happen naturally in a blocked fashion.

3 Parallel Implementation

Since quantum-chemistry calculations are deployed widely, there have been a number of efforts to parallelize such codes [3], [4], [5]. We take a message passing SIMD approach using Fortran 90 and the MPI library to achieve portability and performance. Whenever possible, we call to vendor-provided, optimized library routines.

The most critical decision is about the data layout. A partitioning by vector indices is the best choice [3]. This means each processor holds a certain range of indices of each vector, e.g. processor 0 holds the first 100 vector components of all m vectors, processor 1 holds components 101 through 200 etc. This will make the calculation of dot products between vectors trivial – there is only a global summation required after each processor has computed its local part of the dot product. This generalizes easily to the case of multiple dot products, i.e. matrix-matrix multiplies. We perform the global summation with a call to the MPI reduce function.

How do we assign vector indices to processors? To achieve load balance on the dot products, we should have an equal number of vector indices on each processor. Which ones exactly is unimportant to the dot products, but not to the 3d-FFT.

Figure 1 shows the FFT grid, which is occupied with non-zero components only within a sphere at the center of the grid. The grid points inside the sphere are the vector indices in Fourier space. A inverse 3d-FFT is performed by doing an inverse 1d-FFT along the z direction first, then the y direction, and lastly the x direction. Since at first only data points within the small sphere (its diameter is half the size of the grid) are nonzero, we can perform the first stage inverse 1d-FFT just along the vertical colored lines. To make this operation happen completely local, a processor always holds a complete "line" along the z-direction. In other words, each processor holds vector indices corresponding to complete line segments inside the sphere. Figure 1 shows some of those as dashed lines inside the sphere. To achieve good load balancing, we follow a suggestion by Andrew Canning and hand out the "lines" to the processors such that the processors get a balanced number of short and long lines (the length of a line depends on the place where the line pierces the sphere). The inverse 1d-FFTs are computed by calls to optimized library routines.

After the inverse FFT along z, the data has spread out, and now occupies a cylinder inside the grid (Figure 2). At this point, each processor communicates a single message to all the other processors[1] to achieve a transposition of the grid, and to arrive at the data layout shown in Figure 2. Then the inverse 1d-FFT along the y-direction is a local operation.

[1]The packing of this message is overlapped with communication by using non-blocking MPI send and receive.

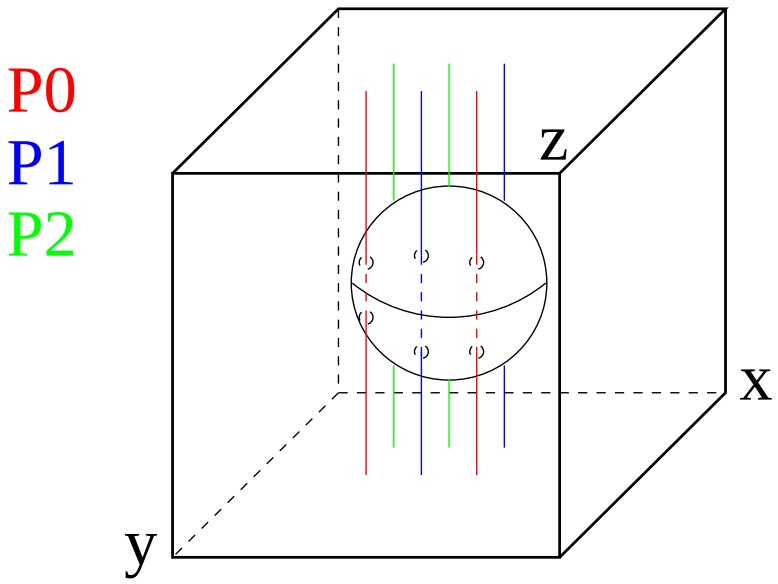

Figure 1: The first stage of the inverse 3d-FFT requires no communication, since the data layout is such that the inverse 1d-FFT along the z-direction is completely local. The grid points inside the sphere correspond to vector indices, those outside are zero at first.

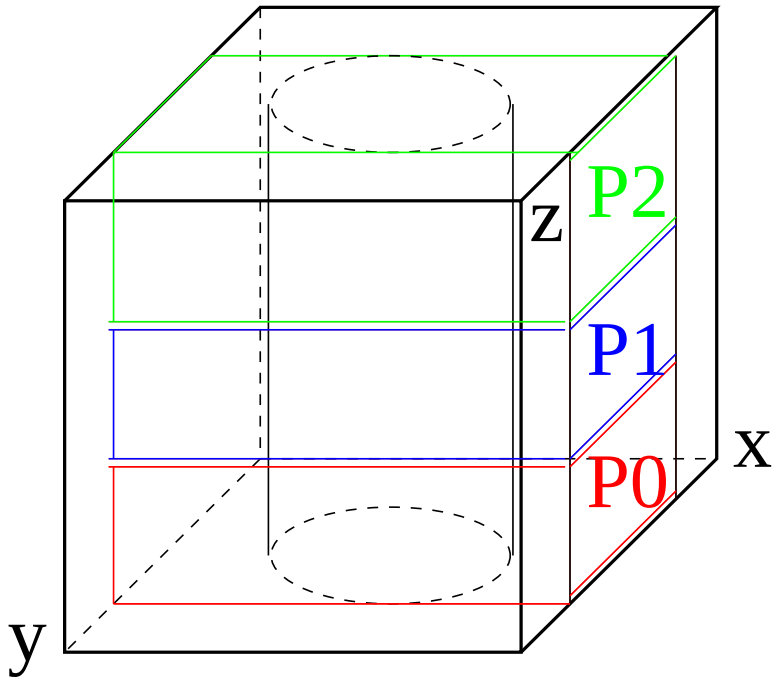

Figure 2: After the first stage of the inverse 3d-FFT, the data has spread out to form a cylinder. A transposition of the grid leads to the data layout shown here, and the inverse 1d-FFT along the y-direction can be done.

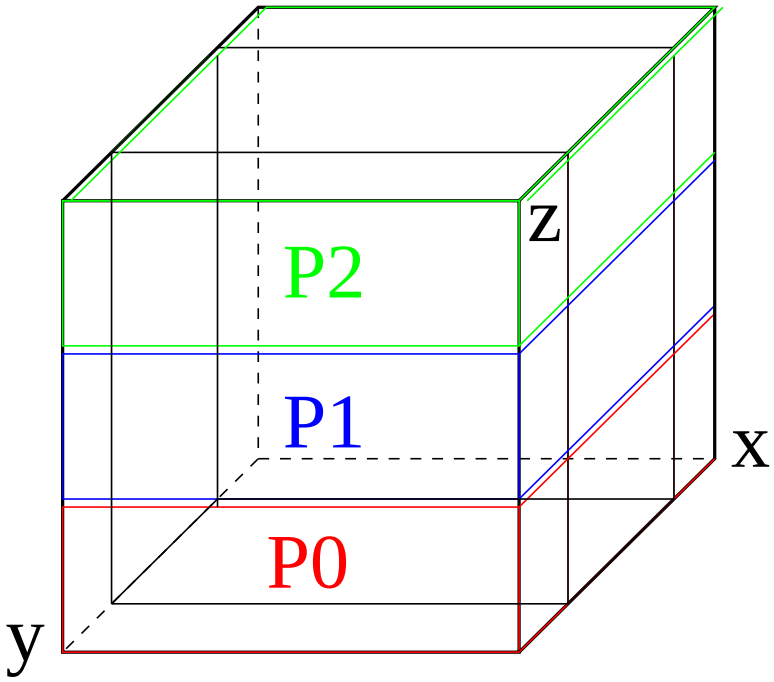

Figure 3: Third stage of the inverse 3d-FFT. After the second stage, all data points within a parallel epiped are nonzero, and a second transpose brings the data into the form shown here. Now, the final inverse 1d-FFT along the x-direction can be done locally.

After the second stage, all data points within a parallel epiped are nonzero, and a second transpose brings the data into the form shown in Figure 3. Now, the final inverse 1d-FFT along the x-direction can be done locally. Since the Fourier grid sizes are preferably powers of two, and so are often the number of processors, the second transpose does not always require communication. In the special case where the grid size in z-direction is an integral multiple of the number of processors, the transpose reduces to a local memory copy.

After the last stage of the inverse 3d-FFT is performed, and the vector is present in realspace, we directly multiply point-by-point with the local potential operator $V_{loc}(\mathbf{r})$, and immediately perform the *forward* 1d-FFT in x-direction, which is the first stage of the forward 3d-FFT that brings the vector back to Fourier space. By combining these operations, we can keep the data in cache as long as possible. The remaining two stages of the forward 3d-FFT are analogous to those of the inverse 3d-FFT, and will not be described here.

We conclude this section by summarizing the merits of hand-coding the 3d-FFT based on 1d-FFTs as opposed to using vendor-provided 3d-FFT routines:

- Not all vendors provide 3d-FFTs with arbitrary initial data layout, but we found 1d-FFTs on all platforms.
- Since initially only a small part of the grid is occupied, we save by not Fourier transforming zeros.
- We avoid a third transposition to bring the Fourier grid back into the original orientation.
- Immediately multiplying with the diagonal local potential operator $V_{loc}(\mathbf{r})$ between the inverse

and the forward FFT in the third stage minimizes the amount of data movement in and out of the cache.

4 Performance measurements

In this section, we present performance numbers for the computational kernels, that is the 3d-FFT and the matrix-matrix multiplies. For any system larger than about 5 atoms, more than 95% of the time is spent inside these kernels. We gauge their speed on our three production platforms: The SGI PowerChallenge, the Cray T3E, and the IBM SP2. All are based on fast, super-scalar RISC processors: The 16 nodes of the PowerChallenge (at NCSA in Illinois) are equipped with 195 MHz MIPS R10000 processors, whereas the 128 nodes of the T3E have a 300 MHz Alpha EV5 chip inside. On the IBM SP2, we use the "thin" nodes with the 66.7 MHz Power2 chip. In contrast to the PowerChallenge, the T3E and SP2 are scalable parallel computers.

As far as the memory hierarchies are concerned, the SGI Power Challenge has a unified 2 MB floating point data cache per node, which is also used as a secondary cache for instructions and integer data. The nodes go through a wide, fast, shared snoopy bus to access the global shared memory. The T3E on the other hand is a distributed memory machine, where the individual nodes have 256 MB of memory, and the EV5 Chip has a 8 KB direct-mapped L1 and a 96 KB 3-way set associative L2 cache. The T3E nodes communicate through a scalable high-bandwidth low-latency interconnect. The IBM also is a distributed memory machine, but has a fat-tree high-performance network connecting its ("thin") nodes. Those have a Power2 chip with a 64KB L1 cache and no secondary cache. In contrast to the "thick" nodes, the bus to local memory is only 64 bits wide.

As a test problem, we took a sample of amorphous hydrogenated carbon with 64 carbon and 12 hydrogen atoms. We use a plane-wave cutoff of 90 Rydbergs which leads to a matrix size of $N = 57000$, and the number of eigenvectors[2] is $m = 21$.

The inverse and forward 3d-FFT involved in applying $\hat{H}$ to a wave function requires a grid of $96 \times 96 \times 96$. The 1d-FFTs are performed by calls to the LIBSCI (Cray), COMPLIB (SGI), and ESSL (IBM) library routines. On all three machines, the native MPI implementations are used.

Figure 4 shows the performance *per node* for a varying number of processors. On the T3E we had to run on 4 or more processors in order to fit the data into memory. Although the T3E at NERSC has currently 128 nodes, only a maximum of 64 are available at the moment, and only the 32 node queues are conveniently accessible. To compute the MFLOPS, the operation count for a 1d-FFT of length p was assumed to be $5p\log_2(p) - 6p + 6$.

All machines fall short of their peak performances of 390 MFLOPS/node (SGI PowerChallenge), 600 MFLOPS/node (Cray T3E), and 266 MFLOPS/node (IBM SP2). The FFTs are very memory-access intensive, and are not expected to perform too well on RISC-based machines where the memory access times are much longer than on vector architectures. Indeed we find them to run at about 76 MFLOPS/node on 4 T3E processors, decreasing gradually due to communication to about 62 MFLOPS/node on 32 processors. On the PowerChallenge, they run at about 48 MFLOPS/node on a single processor. When running on more processors, the required communication first reduces the performance on the PowerChallenge. On 8 or 16 processors though, one can see the effects of the increased aggregate cache, which leads to super-linear speedup when running on 16 nodes (54 MFLOPS/node). The performance curve of the SP2 has a characteristic similar to the one

[2]The number of eigenvalues is lower than what would be really required. This is to allow us to fit the problem into the memory of four T3E nodes

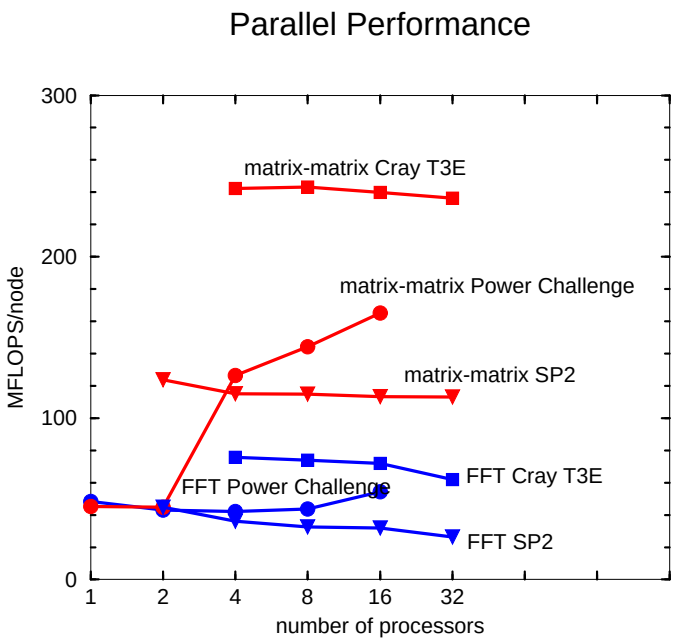

Figure 4: Floating point performance in MFLOPS/node on the Cray T3E, the SGI PowerChallenge, and the IBM SP2 for varying number of processors. The 3d-FFT has a grid size of $96 \times 96 \times 96$. The matrix-matrix multiplies are of size $(21 \times 57000/N_{proc}) * (57000/N_{proc} \times 64)$.

of the T3E. It starts out at about 45 MFLOPS/node on two nodes, and decreases steadily to about 26 for 32 processors. Notice that the absolute performance numbers one would get from a simple benchmark test on a 1d-FFT are much higher, since our number includes not only the communication overhead, but also time-consuming memory copies for the transpose operation and the time to apply the operator $\hat{V}_{loc}$ between the inverse and the forward FFT. The 3d-FFT intrinsically implies a large amount of communication, and requires a parallel supercomputer as opposed to a cluster of workstations.

On the matrix-matrix multiplies, the Cray excels with a performance between 242 MFLOPS/-node (on 4 processors) and 236 MFLOPS/node (on 32 processors). The IBM slows down from 124 MFLOPS/node (2 processors) to 113 MFLOPS/node (32 processors) in a similar fashion. The performance curve of the PowerChallenge has more structure. Like in the case of the 3d-FFT, we see a super-linear speedup for the matrix-matrix multiplies, but this time much more pronounced. This is most likely also cache related. A simple matrix-matrix multiply benchmark reveals that the performance of the PowerChallenge degrades substantially when the shape of the rectangular matrices deviates strongly from the square. This is the case for the matrices multiplied here, because $N/N_{proc} \gg n_{nloc}$ and $N/N_{proc} \gg m$ if N_{proc} is small. Using more processors thus brings the matrices into a more convenient shape. The fact that the Cray T3E and the IBM SP2 are not showing such sensitivity to the shape of the matrix might indicate a superior cache blocking technique of the ZGEMM routine in LIBSCI and ESSL.

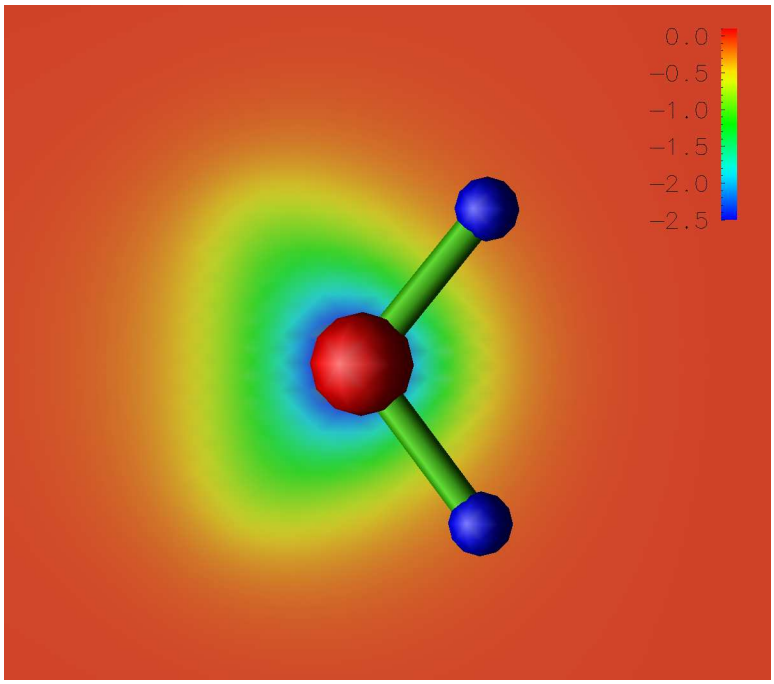

Figure 5: Trace of the susceptibility tensor χ as a function of position in the plane of a water molecule. We were surprised about the fast decay of trace(χ) away from the oxygen atom, and the anisotropy of the distribution caused by the hydrogen atoms.

5 Application

We have tested our method successfully on a number of small molecules, and computed their NMR chemical shift[1]. We are now applying it to larger, more interesting systems. We have performed calculations on ice (12 atoms, matrix size $N = 8500$, number of eigenvalues $m = 16$), a diamond surface (22 atoms, $N = 12600$, $m = 41$), amorphous carbon (76 atoms, $N = 39000$, $m = 134$), and liquid water (96 atoms, $N = 64000$, $m = 128$), all with excellent outcome. Just for the fun of it, we show in Figure 5 a plot of the trace of the susceptibility tensor χ as a function of position in the plane of a water molecule.

Unlike many non-commercial scientific codes, our program is very user friendly, since it has dynamic memory allocation throughout. Also, the data layout is done on the fly and entirely by the program itself, such that parallel runs do not require additional user input. Currently there are about eight active users. The largest calculations (liquid water) typically run within 4-6 hours on 32 processors of the Cray T3E. On the SGI PowerChallenges, we normally run on 16 processors in dedicated mode. Our total group uses about 50000 node hours of parallel CPU time a year. We have accomplished the transition from vector to parallel architectures successfully. The available parallel platforms (T3E, PowerChallenge, SP2) are all stable production machines. In fact, we have not bothered yet to port the code to the vector J90/C90 platforms, although they will certainly perform well on the FFT.

6 Conclusion

We present a parallel implementation of a novel method to compute the NMR chemical shift in condensed matter. We designed the algorithm in a parallel fashion from the very beginning, and could reuse building blocks from our parallel eigensolver. By using a linear systems solver to avoid computing the full spectrum of a large matrix, our scheme becomes suitable for large systems. For the conjugate-gradient linear solver, we suggest a blocking strategy for better cache utilization. The most difficult part of the parallel implementation is a 3d-FFT, which we describe in detail. Fortran 90 and MPI are used to arrive at an efficient, portable and easy-to-use code. We present performance numbers for the computational kernels on our production platforms: Cray T3E, SGI PowerChallenge, and IBM SP2. The good return times, processing power, and memory available on parallel machines has allowed us to compute within a short time frame the chemical shifts of several interesting systems.

References

[1] F. Mauri, B. Pfrommer, and Steven G. Louie, Phys. Rev. Lett. 77, 5300 (1996)

[2] F. Mauri and G. Galli, Phys. Rev. B 50, 4316 (1994)

[3] L.J. Clarke, I. Stich, and M.C. Payne, Computer Physics Communications 72, (1992) 14-28

[4] J.S. Nelson, S.J. Plimpton, and M.P. Sears, Phys. Rev. B 47, (1993)

[5] A. Canning, unpublished.