DOE/ER/14761-1

An MPI Implementation of the MCHF Atomic Structure Package

Mikhail SAPAROV and Charlotte FROESE FISCHER
Department of Computer Science
Box 1679B, Vanderbilt University
Nashville, Tennessee 37235 USA

email: misha@vuse.vanderbilt.edu and cff@vuse.vanderbilt.edu

November , 1996

Prepared for the U.S. Department of Energy
under grant number DE-FG02-97ER14761

Abstract

An existing PVM implementation of the MCHF atomic structure package showed good reliability and efficiency on a cluster of workstations for problems of moderate size. This report describes a revised MPI (Message Passing Interface) version which has removed some of the bottlenecks that arise as the problem size increases. Besides algorithmic improvements, the revised code takes advantage of MPI's industry standard for distributed computing. The performance of the code on several architectures is reported.

1 Introduction

Atomic data is needed in many areas of science and engineering and the properties that can be measured experimentally in a laboratory setting are often severely restricted as to the atoms or ions, the energy regime, or the property. Thus computation becomes an alternative source of data which is checked with experiment, when available, and then applied to systems and properties of interest. With today's powerful workstations and other high-performance platforms, it is possible to improve both the accuracy to which this data can be computed as well as the complexity of the cases that can be considered. One general purpose method that has been used reliably for a variety of problems is the multiconfiguration Hartree-Fock (MCHF) method. The MCHF atomic structure package is a collection of programs for the computation of a variety of atomic properties [1].

2 The nature of the problem

The state of a many-electron system is described by a wave function Ψ that is the solution of a partial differential equation (called the wave equation),

$$(\mathcal{H} - E)\Psi = 0, \tag{1}$$

where $\mathcal{H}$ is the Hamiltonian operator for the system and E the total energy. The operator $\mathcal{H}$ depends on the system (atomic, molecular, solid-state, etc.) as well as the quantum mechanical formalism (non-relativistic, Dirac-Coulomb, or Dirac-Breit, etc.). The MCHF code is based on the the non-relativistic Schrödinger equation for which the Hamiltonian (in atomic units) is

$$\mathcal{H} = -\frac{1}{2}\sum_{i=1}^{N}\left(\nabla_i^2 + \frac{2Z}{r_i}\right) + \sum_{ij}{}' \frac{1}{r_{ij}}. \tag{2}$$

Here Z is the nuclear charge of the atom with N electrons, r_i is the distance of the ith electron from the nucleus, and r_{ij} is the distance between electron i and electron j.

Because of the high dimensionality of the wave equation, approximate methods must be used. A very successful model has been the configuration interaction model in which the wave function, $\Psi_{\gamma LS}$, for a state labeled γLS is written as an expansion of M antisymmetrized configuration state functions (CSF), $\Phi(\gamma_i LS)$. Then

$$\Psi_{\gamma LS}(\{X_j\}) = \sum_{i=1}^{M} c_i \Phi(\gamma_i LS;\ \{\mathbf{r}_j\}), \tag{3}$$

where $\{\mathbf{r}_j\} = \{r_1, \theta_1, \phi_1, \sigma_1, \cdots, r_N, \theta_N, \phi_N, \sigma_N\}$. The r_j, θ_j, ϕ_j are spherical coordinates in three dimensional space, σ_j is the spin-space coordinate for electron j, and γ represents other parameters that are needed for complete specification of the state. Each CSF, in turn, is a linear combination of products of the form,

$$\prod_{j=1}^{N} \frac{1}{r_j} P_{n_j l_j}(r_j) Y_{l_j m_{l_j}}(\theta_j, \phi_j) \chi_{m_{s_j}}(\sigma_j), \tag{4}$$

where the spherical harmonics, Y_{lm_l}, and spinors, χ_{m_s}, are well-known functions. The radial functions, $P_{nl}(r)$, may be known functions or may need to be determined.

By Eq. (1), the total energy of the atom is given by

$$E = \langle \Psi_{\gamma LS} \mid \mathcal{H} \mid \Psi_{\gamma LS} \rangle, \tag{5}$$

assuming $\langle \Psi_{\gamma LS} \mid \Psi_{\gamma LS} \rangle = 1$ [1]. Using Eq. (3) and the multipole expansion for $1/r_{ij}$,

$$\frac{1}{r_{ij}} = \sum_k \frac{r_<^k}{r_>^{k+1}} P^k(\cos\theta), \tag{6}$$

where $r_<$, $r_>$ are the lesser and greater of r_i and r_j, respectively, and $P^k(\cos\theta)$ is a Legendre polynomial in $\cos\theta$ where θ is the angle between $\mathbf{r}_i$ and $\mathbf{r}_j$, the energy may be expressed as

$$\begin{aligned} E &= \sum_{ij} c_i c_j H_{ij} && (7) \\ &= \sum_{ij} c_i c_j \left(\sum_{stuv;k} A^{ij}_{stuv;k} R^k(s,t;u,v) - \frac{1}{2} \sum_{qw} C^{ij}_{qw} L_{qw} \right) && (8) \\ &= \sum_{stuv;k} A_{stuv;k} R^k(s,t;u,v) - \frac{1}{2} \sum_{qw} C_{qw} L_{qw} && (9) \end{aligned}$$

where the R^k are two-dimensional integrals,

$$R^k(s,t;u,v) = \int_0^\infty \int_0^\infty dr dr' P_s(r) P_t(r') \frac{r_<^k}{r_>^{k+1}} P_u(r) P_v(r'), \tag{10}$$

and the L_{qw} are the one-dimensional integrals,

$$L_{qw} = \int_0^\infty dr P_q(r) \left[\frac{d^2}{dr^2} + \frac{2Z}{r} - \frac{l(l+1)}{r^2} \right] P_w(r). \tag{11}$$

The abbreviation, $P_i = P_{n_i l_i}$, has been used here. The $A^{ij}_{stuv;k}$ and C^{ij}_{qw} are called angular coefficients and can be computed using Racah algebra [2]. Because of the cusp in the integrand of Eq. (10), Slater integrals are often evaluated by first solving a pair of first-order differential equations followed by a one-dimensional integral [3]. Thus their evaluation is non-trivial. Eq. (9) represents the energy as a sum of integrals with coefficients that define the contribution a given integral makes to the energy which depends on the expansion coefficients $(c_i, i = 1, \ldots, M)$.

The $M \times M$ symmetric matrix with elements

$$H_{ij} = \langle \Phi(\gamma_i LS) \mid \mathcal{H} \mid \Phi(\gamma_j LS) \rangle, \tag{12}$$

is called the interaction matrix. Applying the variational condition [4] to Eq. (7) and requiring that the energy be stationary with respect to perturbations in the solution (i.e. $\partial E/\partial c_i = 0$ for all i) leads to the matrix eigenvalue problem

$$(H - E)c = 0, \text{ where } H = (H_{ij}). \tag{13}$$

[1] The "bra-ket" notation $\langle\, ..|\mathrm{Op}|..\, \rangle$ implies integration and summation over all variables

Thus the total energy is an eigenvalue of the interaction matrix, and the expansion coefficients of the wave function are the corresponding eigenvector. Of course, in order to compute the interaction matrix, the radial functions need to be known.

In the MCHF method, radial functions are determined that optimize the accuracy of a given expansion. This requires the solution of a set of coupled differential equations, one for each radial function or orbital. The differential equations depend on the expansion coefficients and so both the eigenvalue problem and the system of differential equations need to be satisfied simultaneously.

The MCHF problem is solved iteratively by what is often called the multi-configuration self-consistent field (MC-SCF) method. Using estimates of radial functions, expansion coefficients are determined. Then, in turn, radial functions are updated so as to leave the energy stationary: a new interaction matrix is computed and an improved expansion vector obtained. This process is iterated until results are "self-consistent".

In the MCHF atomic structure package, the NONH program [5] generates the angular coefficients (of Eq. 8) and the associated list of integrals (of Eq. 9), writing the information to a file called a `c.lst`. The MCHF program [6] solves the optimization problem, computing both the radial functions and the mixing coefficients, using the information in the `c.lst`. The data structure of the latter must support the efficient creation of the interaction matrix given the values of a list of integrals, as well as determine how a given integral contributes to the energy. If the radial functions are known, a CI [7] program determines selected eigenvalues and eigenvectors which requires the generation of the interaction matrix.

Once the wave function has been determined, other atomic properties can be predicted as expectation values of appropriate operators, i.e.

$$\langle \text{property} \rangle = \langle \Psi_{i'} | \text{OP} | \Psi_i \rangle, \tag{14}$$

where OP is the operator associated with the property and Ψ_i and $\Psi_{i'}$ are wave functions for the initial and final state, respectively. In some cases, as for the energy, $\Psi_i \equiv \Psi_{i'}$.

Substituting (3) into (13) we then get the result

$$\langle \text{property} \rangle = \sum_{i,i'} c_i c_{i'} \, \langle \Phi(\gamma_i LS) | \text{OP} | \Phi(\gamma_{i'} LS) \rangle. \tag{15}$$

3 Parallel Implementations

Of the programs mentioned above NONH, MCHF, and CI represent the time consuming stages which often also have large memory and/or disk requirements. All parallel implementations todate have been for distributed memory architectures. The first, for the Intel iPSC/2 [8], showed that by distributing the data associated with columns of the matrix in an interleaved fashion, NONH exhibited an almost perfect degree of parallelism, even for as many as 64 nodes. MCHF however, performed poorly. The second implementation, for the Intel iPSC/860 [13], included algorithmic improvements, in that only the lower half of the interaction matrix needed to be stored in a distributed fashion and NONH was required to perform some post-processing that enhanced performance for MCHF. An efficiency of about 50% was achieved with 8 nodes. However, a severe constraint on these platforms was the

lack of any virtual memory. With only 8-16 Mbytes per node, large problems could only be solved with a large number of nodes. Clearly, memory was a scarce resource.

A cluster of workstations, with memories of 64-128Mbytes and 250-500 MBytes of swap space offered attractive alternatives, at the cost of slow ethernet communication. This option was facilitated by PVM (Parallel Virtual Machine) software [14]. Conceptually, PVM consists of software that supports distributed computing on participating UNIX hosts on a network, allowing them to cooperate in a parallel computing environment. A daemon (*pvmd*) is started on each of these hosts and programs communicate between hosts through the local daemons. This distributed communication control eliminates the overhead of a master control process, and facilitates functionalities such as point-to-point data transfers and dynamic process groups. The PVM implementation of the MCHF package, included a change in the matrix eigenvalue solver in the form of sparse matrix methods for the storage of the interaction matrix and the use of Davidson's algorithm [15] for finding a few selected eigenvalues. This change made it possible to increase the problem size from expansions of a few thousand CSFs on the Intel iPSC/860 to about 20,000 CSFs on local workstations and, for a CI calculation which also exhibits a large degree of parallelism, to about 100,000 CSFs on an IBM SP-1 computer using 14 nodes.

4 An MPI implementation

Though the PVM implementation has shown good performance and robustness, there were several motivations for switching to the Message Passing Interface (MPI) [16]. First, MPI is an industry standard, so porting to another workstation or to a Massively Parallel Processor (MPP) platform should be relatively simple. Since MPI is a standard, vendors often provide their own library as a part of the parallel environment. Second, MPI unlike PVM, uses an abstract communication device(ADI) on which a high level application programmer interface (API) is implemented. The API implementation consists of an entirely portable code and does not depend on the implementation of ADI. At the same time the dependencies on the low-level transport level are combined in an implementation of ADI which is created by MPP and workstation network vendors to get maximum performance of the underlying hardware. This structure does not give much improvement in the case of a workstation cluster connected via Ethernet since it is an architecture on which PVM is based, but it gives the possibility of creating a portable high level code which gives maximum performance on any possible hardware.

Converting from PVM to MPI is not difficult since both libraries have a similar structure. The major difference between the packages is the concept of dynamic groups in PVM versus static groups in MPI. However, since dynamic groups were not used in the MCHF codes, that issue did not arise. A big difference is that in PVM data needs to be packaged into a buffer before sending, then it is sent using a general send method, and later unpackaged on the receiving end, while in MPI the send method provides a datatype argument which matches PVM's use of different packaging routines for different types of data. This MPI feature releases the extra overhead arising from coding of the data during the packaging, which is not necessary in the case of homogeneous nodes. The process startup also varies. In PVM, first a master PVM daemon is started from the UNIX shell. Then

the first instance of the program (master) is started on the same node as the daemon, which starts the other processes via a `pvmfspawn()` call. In contrast, the MPI standard has not specified the mechanism for process startup, and hence it depends on a platform dependent implementation. The only guarantee is that a copy of the executable will be started simultaneously on each node of a parallel pool. The chosen MPI implementation, MPICH, contains a script to simplify startup under different environments. On the other hand, on the IBM SP2, MPI is a part of the IBM Parallel Environment(PE), and a program starts as a regular PE executable. In general, conversion was straight forward, but the MPI suite of routines gave extra opportunities for making the code easier to understand, more standardized, and optimized.

5 Benchmarks for Evaluating Performance

The problem size and resource utilization depends on several parameters:

1. M, the size of the matrix (or the number of CSFs). However, the number of matrix elements is proportional to M^2 even in the case of sparse matrices. The amount of data generated and stored in the `c.lst` also grows roughly as M^2
2. N, the number of radial functions or orbitals. This determines the the number of integrals which grows somewhere between N^4 and N^5 and also the number of systems of differential equations that need to be solved.
3. L, the length of the `c.lst`.

Most of the MCHF benchmarks mentioned in this paper were done on 2 cases:

1. This was a calculation for $\mathrm{O}^- \ {}^2P$, with $N = 15$, $M = 14,000$, and L about 208MB. The number of iterations required by the Davidson algorithm were 45 for the first call and 29 for the second.
2. This was a calculation for $\mathrm{C}^- \ {}^4S$, with $N = 42$, $M = 20,311$, and L about 452MB. The number of Davidson iterations were 42 for the first call and 26 for the second.

Thus the second benchmark has about twice the disk I/O and considerably more computation during the SCF process of MCHF.

Also four different hardware platforms were used : 4 IBM RS6K/370, 4 DEC ALPHAstations 200 machines, an IBM SP2 cluster, and a CRAY T3E. All nodes in the IBM workstation cluster have the same hardware configuration: 128 MB of RAM, 256 MB of virtual memory, 1GB local disk, and ethernet connection. While all IBM's were completely identical, the DEC's varied slightly; two nodes had 96 MB of RAM and two only 64MB. However, all nodes had 9GB local disk (though the group quota was between 1-2 GB), and all DEC's had 256 MB of virtual memory. They were connected via a fast ethernet switch for point-to-point communication (which permits parallel communication), though the implementation of MPI for DEC nodes does not have the capability for asynchronous operations due to remaining software bugs, so only the synchronous version of the code was used. The IBM SP2 cluster consisted of 16 TH2 66MHz nodes with 256 MB of RAM,

Table 1: Timing information, speed-up, and CPU utilization for NONH on an IBM RS6K/370 for the second benchmark.

# of nodes	time(min)	speed-up	CPU util.
2	40.09		99%
4	20.52	1.95	99%

512 MB of virtual memory, 1GB local disks, interconnected through High Performance Switch(HPS). The CRAY T3E has 128 DEC ALPHA 300 nodes with 256 MB of RAM per node, 370GB of the disk space. The nodes are interconnected by a high-bandwidth network arranged in a 3D torus.

6 Performance enhancement and evaluation

Performance is monitored through speed-up (s), efficiency (e), and scalability. Let t_i be the time required to perform a task on i nodes, then

$$s = t_1/t_n; \quad e = t_1/(nt_n).$$

Thus efficiency is a measure of how effectively processor utilization is maintained during parallel execution of a task as more nodes are used. Scalability implies that a given efficiency can be maintained as the problem size and the number of processors grow [9]. These criteria will be used to analyze and evaluate performance.

Three programs were considered: NONH, MCHF and CI. For the programs NONH and CI, time was measured for the entire run, whereas for MCHF, only two iterations were analyzed.

6.1 NONH

Both NONH and CI are logically simple from the point of view of parallelism. For the former, each node generates data for the columns of the matrix assigned to it, then sorts the data it produced, and, at the end, sends a small amount of information to the master node. The interleaved distribution is surprisingly well balanced for large cases. The near perfect parallelism is shown in Table 1. Because the problem was too large for a single node, speed-up was computed relative to a two node calculation.

6.2 CI

The CI program also generates data for its assigned columns, but in this case it assembles the interaction matrix using a list of integrals. Once all data has been generated, a barrier point is reached, and all nodes begin to participate in the computation of a few eigenvalues/eigenvectors using the Davidson algorithm. The latter requires a global summation during each iteration, but since the time required for the generation of the matrix far exceeds the time required to find the eigenpairs, the overall performance of this program is also excellent.

CI also has shown little dependency on problem size in terms of CPU usage for the cases considered to date. The process of generating the matrix requires a list of radial integrals. In the previous implementation each node generated its own integrals so there was extensive duplication in the evaluation of integrals. However, it was found that for current problems, there were as many as 8,000,000 two dimensional integrals whose evaluation was taking 1-2 hours on workstations. This performance became disturbing particularly when MPP calculations are contemplated. In the new MPI implementation, a global list of integrals was introduced and the task of evaluating the integrals was also distributed, requiring a global "collect". This would improve parallelism (efficiency) but in large cases, using only a few nodes, it is again a relatively small part of the overall task.

A timing experiment on the SP2 showed a speedup of the calculations by up to 20% for big cases(25,860 CSF, 56 orbitals)(fig. 1). Another set of test runs was conducted only for the new code on the T3E machine to see how the application scales on the large number of nodes(fig. 1)

6.3 MCHF

The MCHF program is the one that has caused the most degradation of performance with the growth in task size. Large calculations were tried that lead to bottlenecks in disk I/O and virtual memory processes. An attempt to create an all memory version of the code in order to avoid I/O bottlenecks caused by reading the long `c.lst` files had been done. However, the experiment showed that an inadequate amount of physical memory on the nodes makes it impossible to avoid storage of coefficient lists in swap space, so the only solution was to improve I/O operations somehow. Another visible problem was low CPU utilization even when no I/O operations were executed. Measurements showed that after 2-3 MCHF iterations, the operating system would call a junk a collector in order to allocate contiguous blocks of memory. Since the program used not only physical memory, but also a large amount of virtual memory, the latter operation sometimes took several minutes on a workstation cluster. The disk I/O bottleneck caused an effect of "data starving," where the CPU is idle for long periods of time waiting for the completion of disk I/O. Data starving occurred during the matrix creation and coefficient update phases.

One additional problem with the MCHF code is that it is not nearly as perfectly parallel as the other codes mentioned, because it contains many synchronization points and, in order to avoid even more communication, parts of the calculation are duplicated on every node. This leads to a worse scalability of the application especially with the increase in the number of nodes. MCHF has shown low efficiency and even performance degradation on more than 8 nodes which has restricted expansion to a larger number of nodes. However, removing existing bottlenecks should increase code scalability.

The MCHF code is logically divided into four phases:

1. Matrix creation phase (MAT). This is the part where each node reads the coefficient list in its file `c.lst.nn`, generates its local matrix, and keeps a record of its diagonal elements of the matrix. At the end of the phase, there is global communication in order to collect the diagonal elements among the nodes. This phase may be characterized as I/O intensive.

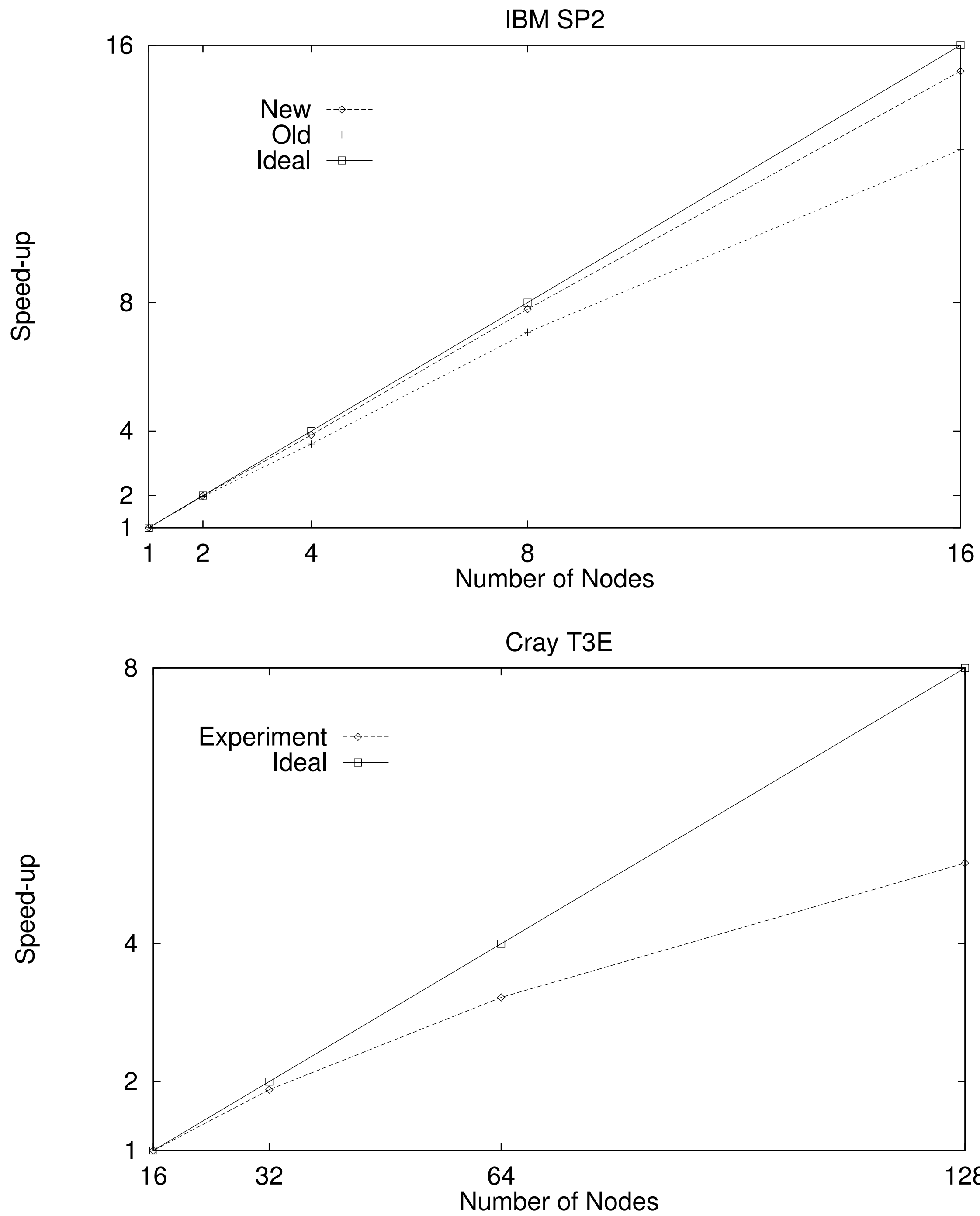


Figure 1: The CI speed-up on the SP2 and T3E for the new distribution algorithm with a matrix size of $M = 25860$ and $N = 56$.

Table 2: MPI times with and without the alloc/dealloc on each iteration, for Benchmark 2 and 10 MCHF iterations.

version	time(min)	CPU util.
original	204.812	24.41%
improved	165.726	30.17%

2. Davidson iteration phase (DVD). Every iteration involves a sparse matrix vector multiplication in memory. Each node has its own copy of the vector (a limitation in the case of MPP) and first computes its contribution to the result vector which is followed by a global summation, requiring communication. This phase may be characterized as computationally intensive but with one synchronization point within every iteration [17].

3. Self-consistent field iterations (SCF). In this phase, the system of coupled differential equations are solved, one for each orbital. For each differential equation, each node computes its contribution to a function, say $Y(r)$, which is then formed by a global summation. Similarly, each node computes its contribution to a function, say $X(r)$, which also is then summed globally. This part is computationally intensive with frequent communication though the length of the vectors in the global summation is relatively short as compared with those of the DVD phase.

4. Coefficient update phase (UPC). In this part the list of coefficients is being read from disk and some computation is performed to form the coefficients of Eq. (9) from those of Eq. (8), which are stored in the file. This part is I/O intensive.

6.3.1 Improvements

The following steps have been implemented in order to improve the MCHF performance:

1. Both PVM and MPI versions of the MCHF code were slightly redesigned for better memory allocation in order to remove bottleneck caused by a junk collector call. In order to avoid this situation, allocation/deallocation calls were minimized. In the earlier version memory for matrices had been allocated just prior to call to the Davidson subroutine and released after completion. This solution was implemented in order not to keep extra memory between the phases when it was not needed. However, careful timing analysis showed that extra memory allocation does not cause as much code performance degradation as the junk collector call and the following reallocation of virtual memory. A new solution was to allocate memory only once during the first MCHF iteration and deallocate only after the completion of the last iteration. This change gave 6% increase in the CPU utilization (Table 2).

2. As was mentioned earlier, the effect of data starving appeared in large cases during the matrix creation part of the MCHF code. The creation of the matrix consists of 3 easy divisible parts: reading coefficients from the disk, calculating local matrix, collecting the diagonal matrix elements from all nodes. Although the program does I/O in chunks, long

Table 3: A Comparison of times with and without asynchronous reading for Benchmark 2 with 10 iterations on the IBM RS6K/370 and four nodes.

version	time(min)	CPU util.
synchronous	165.726	30.17%
asynchronous	146.102	34.22%

coefficient lists still cause data starvation. However, if the underlying hardware has the ability for asynchronous, interrupt driven communications, we can do disk I/O (reading of the coefficient list) and communication at the same time. The algorithm is rather intuitive:

a) read the chunk of data

b) post an asynchronous receive of the diagonal element

c) calculate the local elements and a diagonal element

d) post an asynchronous send of the diagonal element and go to a)

Thus, if the hardware is capable of doing so, we can have disk I/O and node communication in parallel which gives us, even in the case of a workstation cluster, some improvement in the CPU utilization (Table 3).

3. The previous PVM code contained customized procedures for global and collective operations. These procedures were written for use on workstation clusters connected via ethernet and used sequential algorithms [10] more appropriate for that environment. However, on MPP platforms, different kinds of algorithms, like fan-in, may be more efficient depending on the intercommunication implementation. MPI contains the necessary operational set in standard implementation so, for the sake of portability, the code has been changed to use corresponding MPI subroutines. Moreover, as was mentioned earlier, MPI leaves the communication part, as a part of the ADI, to the vendor implementation so that code gives the best performance for every architecture.

MPICH, the implementation that was used on workstations, uses sequential or tree algorithms depending on the number of nodes in the parallel pool. Timing studies have been done in order to insure that code for global operations performs sufficiently well. Particular interest concerns the global sum(GDSUM in the PVM implementation) operation which is heavily used in the MCHF code. The experiments showed that the long size of a vector in GDSUM can cause extra buffering. Also programs become more vulnerable to a network congestion. The decision was made to not rely completely on the internal library implementation but to partition long messages into relatively small chunks before calling the MPI subroutine, MPI_ALLREDUCE. The size of the chunks should be customized depending on the architecture and network to reach a maximum improvement. The size of 32K has been chosen for workstations connected via ethernet which relies on the benchmarks done by Nevin[11]. If an MPI implementation gives equal or better performance without custom chunking of the vector, the chunk size should be made equal to the largest integer(or any other number larger than vector size). Then the code will not cut the vector before the call. A simple program TEST was developed in order to check the performance for a particular architecture. The program benchmarks the GDSUM operation on various vector sizes under two conditions. First, memory is allocated only on a need basis and then the

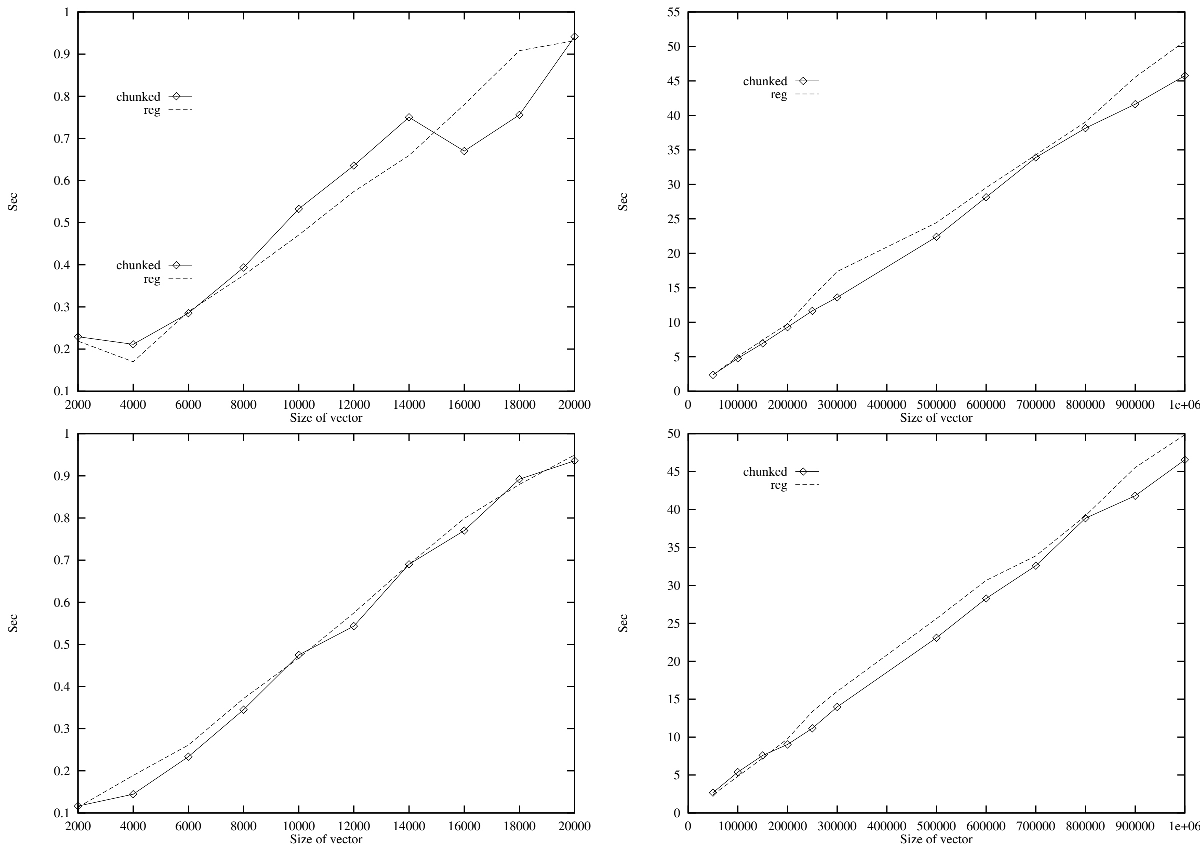


Figure 2: Benchmark of GDSUM using TEST program for IBM RS6K/370. Top pictures represent case without memory allocation, bottom with additional 64 MB allocated

program allocates an extra 64MB of memory before calling GDSUM in order to simulate conditions of the large case MCHF calculations. The corresponding graphs of the results of TEST program for 4 workstations IBM RS6K 370 are on the Figure 2. The results show that for this particular architecture, the chunked version is slightly faster for large cases while it shows similar performance for small vector sizes.

7 Profiling

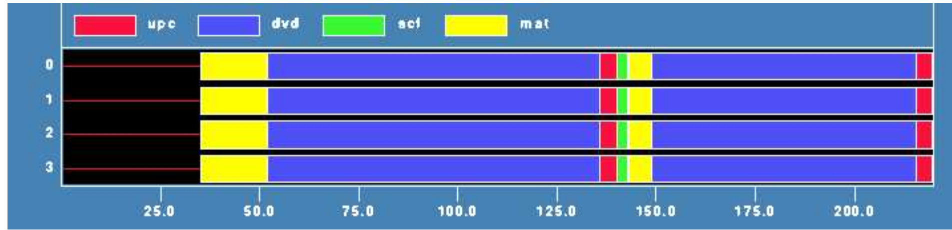
Another advantage of the MPI implementation (MPICH) was an included profiling library with a graphical interface. It made it possible to capture pictures of the time for different stages of the MCHF execution, which helped to understand the underlying processes and for making further conclusions. Some explanation is necessary. Since SCF is believed to be the least scalable, it was interesting to measure the code performance. The duration of the SCF part depends mainly on the number of orbitals, while other phases depend more on number of configurations, so Benchmark 1 with a small number of orbitals but a large M was selected. The behavior of the code in that situation is shown in Figure 3, case i) for the IBM RS6K/370's and ii) for the DEC Alphas. Cases iii) and iv) show the behavior of the code when both M and N are large as in Benchmark 2, for the same architectures,

respectively. Each diagram displays the amount of time in each phase of an MCHF run for the first two iterations. The time scale is not exactly the same but the total amount of execution time is similar: 225 seconds versus 190 seconds for Benchmark 1, and 310 seconds versus 350 seconds for Benchmark 2. It is clear, however, that the distribution of time among the different phases is quite different for the two architectures.

We can see from Figure 3 that although the DEC machines should execute faster according to their MFLOP rate and fast ethernet switch versus the regular ethernet for the IBM's, the times are really quite similar. The reason is partly due to resource imbalance on the DEC Alpha machines. As was mentioned earlier, the code uses virtual memory intensively in large cases. The 80% usage of the physical memory on a machine with 96MB means that we are already using swap space on a machine with 64MB. Thus, in the best case, all nodes are running at the speed of the slowest. In reality, when a program goes into a wait state at the synchronization point, it might be dumped to the swap space by an operating system, which causes extra overhead and gives even poorer performance. Figures 3 iii) and iv) show the difference in behavior between the IBM's and the DEC's for the larger calculation. While the IBM nodes spend the majority of their time in the MAT and SCF parts which are not not particularly memory intensive, on DEC's these phases are faster as expected, but the DVD part takes much longer than on IBM's. It occurs because of a resource imbalance and the smaller RAM on the DEC nodes. The size of the interaction matrix in the second benchmark is 102MB (12.75 MWords (double precision)) per node, and thus exceeds the size of physical memory on the DEC's and causes the virtual memory device to become a bottleneck. The load imbalance worsens in this case since the slowdown caused by the memory bottleneck has exponential distribution, so that machines with smaller RAM lose their speed more. The possible solution in that case is to allocate columns unevenly so that nodes with less memory also are assigned a smaller part of the task. This strategy was investigated by Stathopoulos and Ynnerman [10] to accommodate processors with different speeds.

An essential part of the DVD phase is the global summation. Figure 4 demonstrates synchronization of the GDSUM calls during one Davidson call on both IBM's and DEC's for the smaller case 1. On the DEC, there are 10 iterations in 12 seconds, whereas the IBM requires about 12 seconds for 8 iterations. We can notice lack of synchronization on DEC's by observing that while the nodes with more memory are in the wait state, two others continue their computation. Although according to the performance charts[12] the DEC Alpha 200 is supposed to show approximately 1.5 times better CPU performance than IBM RS6K/370, in our benchmark only the nodes with more memory show the expected speed while on the smaller DEC nodes the time of calculation of one Davidson iteration was approximately the same as on IBM's. It is also noticeable that communication time on the DEC machines is shorter because of the fast ethernet switch.

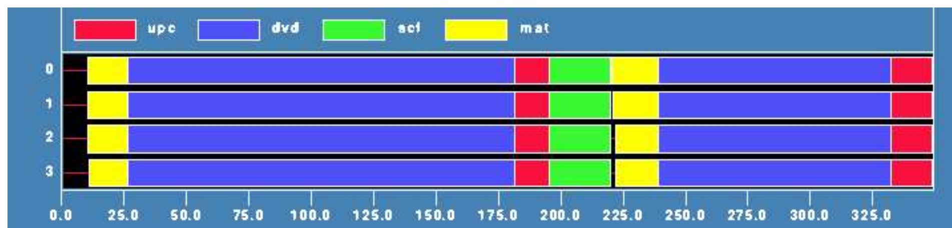
The output of the UNIX command vmstat demonstrates node utilization on the different stages of the MCHF code (Figure 5). These experiments have shown that the previous bottlenecks are partly removed: there is some CPU activity during disk I/O; the CPU utilization during the DVD phase is around 70% on average vs. 60% in the PVM version, there is good CPU utilization during the SCF phase but it is known to lack the parallelism needed for an MPP platform.



i) Benchmark 1 on four IBM RS6K/370's

ii) Benchmark 1 on four DEC Alphas

iii) Benchmark 2 on four IBM RS6K/370's



iv) Benchmark 2 on four DEC Alphas

Figure 3: Time diagrams showing the time distribution in the four phases of an MCHF iteration for two iterations of Benchmarks 1 and 2, on four IBM RS6K/370 workstation and four DEC Alpha workstations. The four phases are: MAT (yellow) for creating the matrix; DVD (blue) for finding an eigenpair; UPC (red) for updating the integral coefficients; SCF (green) for updating the radial functions. The initial set-up is depicted in black.

Figure 4: A comparison of computation (Calc) and global summation (GDSUM) during Davidson iterations: top IBM, bottom DEC.

Table 4: A comparison of execution times and CPU utilization of PVM and MPI versions of MCHF, Benchmark 2.

version	time(min)	CPU util.
PVM	166.814	30.06%
MPI	146.102	34.22%

7.1 Performance of revised code versus previous code

In order to compare the new MPI code with the earlier PVM one the same cases have been run using the PVM version of the code, though with similar algorithms. The runs consisted of 10 iterations on 4 nodes. The changes mentioned earlier concerning alloc/dealloc were applied to the PVM code as well in order to make the comparison unbiased. The execution times and CPU utilization are reported in Table 4. Benchmarks have shown that for large cases the MPI version is faster than PVM. It is connected with the improved (chunked) GDSUM, asynchronization of the matrix creation phase, and all-in-all slightly better performance of the MPI libraries.

8 Supercomputers : the IBM SP2 cluster and CRAY T3E

For the supercomputers, the picture is different. These machines have hardware support for an interrupt driven communication mode, and also fully independent disk I/O and communication processes. On the SP2 nodes communicate via High Performance Switch

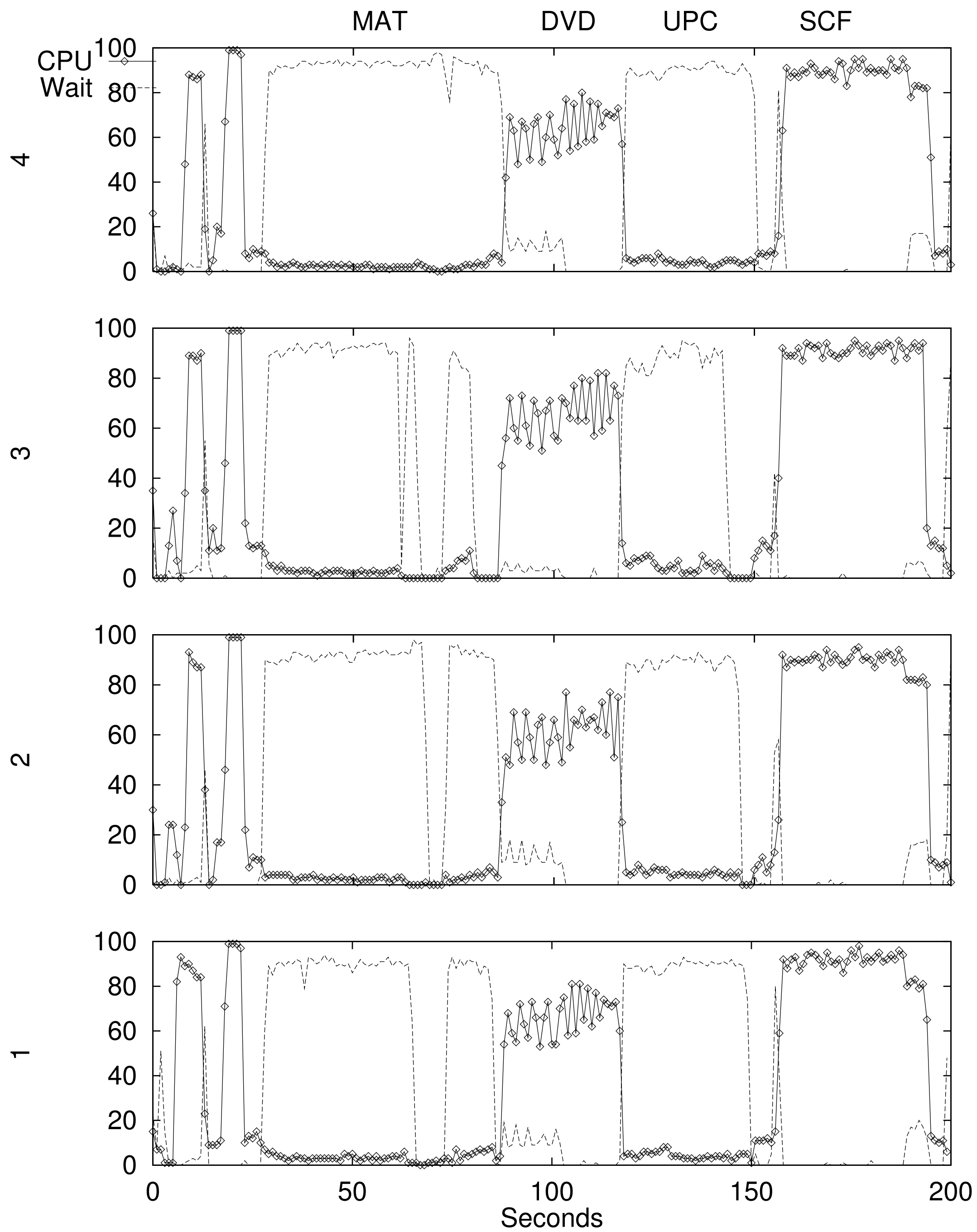


Figure 5: Percentage of time distributions on each processor for CPU, I/O Wait, and Idle time, for an MCHF iteration of Benchmark 2 on four IBM RS6K workstations.

(HPS) while on the T3E through the low latency, high-bandwidth interconnected network. Also there are more nodes which gives different demands on scalability. Timing studies have been done for the CI, NONH, and MCHF code. As expected from the earlier studies on the PVM code [10], CI and NONH showed good scalability in that environment (see Figure 1).

The situation with the MCHF code was different. It was known to scale badly and even to demonstrate performance degradation when the number of nodes was greater than 8. Additional timing studies were done in order to figure out the root of the problem. The MCHF benchmark unveiled an interesting pattern(tables 5,6). Although only the SCF phase scales badly with an increase of number of nodes, it significantly impacts the total speedup, a good illustration of Amdahl's law. Anyway, the MPI code outperforms the previous PVM code which has given performance degradation after 8-nodes, while the current code gives a slow increase. This is connected with better scaling of the MAT phase in the MPI implementation and changes in hardware characteristics of the high performance switch (decrease of the latency time and increase of the communication speed). We can also note that MAT phase scales not so perfectly on the CRAY T3E and for 32 nodes time even increases comparing with 16 nodes. This anomaly may be explained with the fact that while on the IBM SP2 each node has its own disk, on the T3E all processor elements share the same disk subsystem and perform I/O through a 4-Gigabit ring which slows down I/O with the increase of the number of executing nodes due to the resource saturation. Another difference is in the execution environment. On the SP2, the tests were run in the dedicated mode, while on the T3E they were executed from the batch queue and shared the I/O resources with the other jobs in the system. So if some other job was using the disk subsystem intensively, it introduced significant delay into the MAT phase and made scaling look worse like in our case.

The hardware support for asynchronous communication and the use of the interrupt driven mode in IBM's version of MPI made it possible to fully employ the corresponding MPI features like asynchronous SEND and RECEIVE operations. In such environment the use of the asynchronous communication in the MAT phase lead to 1.7 factor in speedup versus the synchronous version for the larger benchmark:

version	time (sec)(per node)	%CPU util
sync	2.85	10
async	1.64	48

9 Conclusions

Removing some of the bottlenecks in the MCHF code reduces the duration of time needed to solve large tasks. Converting PVM code to MPI has been done to keep the MCHF package in compliance with industry standards, making it more portable and improving communication performance. Implemented changes gave a significant improvement to the large calculations. Even a 5-10 percent increase in a CPU utilization reduces computational time in hours for such calculations, which improves reliability and saves resources. The MPI profiling libraries have provided an opportunity to look at the MCHF code "inside", to show

Table 5: The MCHF speed-up by phases on the IBM SP2 cluster. All times are in seconds. Speedup is based on the 2-node time.The abbreviations in the table are as follows: T - total time per node; S - total speedup; E - total efficiency

N	MAT		DVD		SCF		T	S	E
	T	S	T	S	T	S			
2	11.8101		55.2668		71.0587		449.8371		
4	6.0824	1.94	30.1824	1.83	48.3528	1.47	295.0209	1.52	0.76
8	3.1431	3.76	16.5192	3.34	34.4395	2.06	187.5681	2.39	0.60
16	1.6372	7.21	9.1774	6.02	30.2845	2.34	137.2419	3.27	0.41

Table 6: The MCHF speed-up by phases on the CRAY T3E cluster. All times are in seconds. Speedup is based on the 2-node time.The abbreviations in the table are as follows: T - total time per node; S - total speedup; E - total efficiency

N	MAT		DVD		SCF		T	S	E
	T	S	T	S	T	S			
2	40.9208		79.1601		99.7417		627.9755		
4	22.1404	1.85	42.9140	1.84	65.9362	1.51	399.9080	1.57	0.79
8	17.2202	2.38	22.9941	3.44	49.5143	2.01	358.6612	1.75	0.44
16	8.6493	4.73	13.0736	6.05	38.8130	2.57	180.6757	3.48	0.44
32	10.6950	3.83	8.3699	9.46	31.9091	3.12	102.4189	6.13	0.38
64	5.0420	8.12	5.8443	13.54	20.3409	4.90	67.6088	9.29	0.29

time distribution of the MCHF phases in easy to understand pictures, and to perform a more careful analysis of the code.

10 Acknoledgement

This research was supported by the Division of Chemical Sciences, Office of Basic Energy Sciences, Office of Energy Research, U.S. Department of Energy.

References

[1] Fischer, C.F. *The MCHF atomic-structure package.* Computer Physics Communications, 64, 369-398, 1991

[2] Fano,U. and Racah, G.. *Irreducable Tensorial Sets.* Academic Press, 1959

[3] Fischer, C.F. *Self-consistent-field (SCF) and multiconfiguration (MC) hartree-fock (HF) methods in atomic calculations: Numerical Integration approaches.* Comput. Phys. Repts., 3, 273, 1986

[4] Fischer, C.F. *The Hartree-Fock Method for Atoms: A numerical approach.* J. Wiley & Sons, New York, 1977

[5] Hibert, A. and Fischer, C.F. *A program for computing angular integrals of the non-relativistic hamiltonian with non-orthogonal orbitals.* Comput. Phys. Commun., 64, 417, 1991

[6] Fischer, C.F. and Jönsson, P. *MCHF calculations for atomic properties* Computer Physics Communications, 84, 37-58, 1994

[7] Fischer C.F. *A configuration interaction program* Comput. Phys. Commun., 64, 473, 1991

[8] Bentley, M. and Fischer. C.F. *Hypercube conversion of serial codes for atomic structure calculations.* Parallel Computing 18, 1023-1031, 1992

[9] Foster, I. *Designing and Building Parallel Programs* Adison Wesley, New York, 1995

[10] Stathopoulos, A., Ynnerman, A. and Fischer, C.F. *A PVM Implementation of the MCHF Atomic Structure Package.* International Journal of Supercomputer Applications and High Performance Computing, 10, 41-61, 1996

[11] Nevin, N. *The performance of LAM 6.0 and MPICH 1.0.12 on a Workstation Cluster.* Ohio Supercomputer Center Technical Report OSC-TR-1996-4, 1996

[12] Dongarra, J.J. *Performance of Various Computers Using Standard Linear Equation Software.* University of Tennessee, Knoxville, Technical Report CS-89-85, 1996

[13] Fischer, C.F. et al. *A distributed-memory implementation of the mchf atomic structure package.* The Journal of Supercomputing, 8, 117, 1994

[14] Geist, A. et al. *PVM 3 user's guide and reference manual.* Technical report TM-12187, ORNL, 1993

[15] Davidson, E.R. *The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices.* J. Comput. Phys., 17, 87, 1975

[16] Gropp, W., Lusk, E. *User's Guide for mpich, a Portable Implementation of MPI.* Technical Report ANL/MCS-TM-000, ANL, 1995

[17] Stathopoulos, A. and Fischer, C.F. *Reducing synchronization on the parallel davidson method for the large,sparse,eigenvalue problem.* Proceedings of Supercomputing '93 Conference, 172, ACM Press, Portland Kaufmann, 1993