Performance optimization of electronic structure codes on the Cray T3E *

Andreas Stathopoulos[†] Barry Rackner[‡] Yousef Saad[†] James R. Chelikowsky[§]

April 15, 1997

Predicting the electronic structure of complex systems from *ab-initio* principles is an outstanding problem of condensed matter physics. In many of the corresponding calculations the dynamics of the particles are followed until a minimum energy state is found. Central to this minimization problem is the repeated solution of a large, symmetric eigenvalue problem.

Improving on previous methods, our approach is based on high-order finite difference schemes. The resulting matrices are large and sparse, and the number of eigenvalues and eigenvectors required is proportional to the number of atoms in the system. Thus, for complex systems hundreds or thousands of eigenpairs are required. To meet some of the huge computational demands posed by these calculations, we solve the problem only in non-trivial subregions of the domain, and we use a preconditioned eigenvalue iterative solver which can tackle large numbers of eigenpairs. The matrix is not stored and both matrix-vector multiplication and preconditioning operations are performed on the grid stencil. The unstructured domain regions are handled with special data structures that determine the sparsity pattern.

Parallel computing plays also a crucial role in meeting the above challenges. The code has been implemented on the Cray T3D and T3E computers using the Cray C90 as a front-end. A combination of PVM and MPI communication libraries has been necessary for an efficient implementation of the master-slave paradigm. In this combination, all communication within the multiprocessor is performed through the native T3D/E MPI which is faster than the native PVM, while communication with the front-end uses the more flexible PVM library. Despite the unstructured nature of the problem, our benchmarks show scalability of 85% on 64 nodes even for average size cases.

The migration from the C90 parallel vector processor to the cache-based multiprocessor systems T3D and T3E, has created several performance shortcomings. In this paper we describe techniques and optimizations that improve the single-node efficiency of the code.

The processors on both T3D and T3E have a data cache of 8 Kbytes which is directly mapped to the local memory of each processor. In addition, the T3E has a second level set-associative cache of 96 Kbytes and six stream buffers which allow concurrent communication between the second level cache and the local memory. With a 300 MHz clock, the T3E has a peak speed of 600 MFLOPS. However, the observed speeds in our initial implementation were less than one tenth of the peak rate.

Several reasons account for the above behavior. First, a floating point operation is completed in 4 clock periods. Even if the cache and prefetching mechanisms hide all memory latencies, only

* Work supported by NSF grants DMR–9217287, ASC 95-04038, and by the Minnesota Supercomputer Institute
† Department of Computer Science, University of Minnesota
‡ Cray Research
§ Department of Chemical Engineering, University of Minnesota

a fourth of the peak rate can be achieved unless the compiler can schedule multiple operations simultaneously. Second, the concurrent operation of the stream buffers requires an appropriate number of input/output operands. Too many operands cause stream thrashing, while too few underutilize the system. Finally, degradation is caused by the indirect addressing used by the sparse data structures, and the resulting cache misses.

Our first optimization involved a modification of the data structures. Because of the unstructured sparsity pattern, the initial code used a three dimensional index array to specify which points in the domain are considered in the stencil.

```
                row number in the matrix if point (i,j,k) is considered
index(i,j,k) =
                special out-of-matrix index if not considered
```

These indices were used in matrix-vector multiplication and preconditioning operations to multiply a matrix element with the corresponding entry of a vector `x`: `x( index(i,j,k) )`.

There are two problems with this approach: First, `index` must be duplicated on every processor which poses large memory requirements. Second, the neighboring points in the stencil (e.g., `index(i,j+1,k)`, `index(i,j,k-1)`) display no spatial memory locality. In addition to the cache misses occurring when accessing the neighbors in the stencil, the access of `x` has no regular pattern either.

To face the above problems, we dispense with the array `index`, and introduce a two dimensional array `JA` which holds the stencil neighbors for each of the points considered (matrix rows):

```
            row number of the i stencil-neighbor of row j
JA(i, j) =
            special out-of-matrix index if i neighbor not considered
```

The JA array can be viewed as a special case of the Compressed Sparse Row format. Since only the matrix rows are involved in the long (second) dimension of `JA`, different processors may keep only the part of `JA` associated with their local rows. Thus, duplication is avoided and storage scalability is achieved. The short dimension is chosen as the first one, for saving floating point operations while retaining access locality. In this way, `JA` is accessed sequentially, and if a stream buffer can be sustained, the only cache misses occur because of the irregular access pattern of `x`. Overall the use of `JA` reduced the execution time by a factor of 1.5.

Besides the stencil tensor, the matrix also involves a summation of a number of sparse rank one updates. Similar modifications of the corresponding data structures are performed to optimize the memory access patterns. Finally, gather-scatter strategies are considered.

At every step of the iterative algorithm, a vector is orthogonalized to all the previously converged eigenvectors. As the number of the required eigenvectors increases, the orthogonalization becomes the dominant part of the computation. This is performed through the BLAS-2 kernel SGEMV. Initially, when debugging the code for performance on a smaller case, the aggressive compiler optimization for stream buffers was turned off since it degraded performance. When running larger cases the lack of good stream optimization became apparent, with SGEMV performing at about 56 MFLOPS even for the largest cases. Switching stream optimization on raised SGEMV performance above 100 MFLOPS, with the large cases reaching 148 MFLOPS. The whole program received a twofold speedup.

The above optimizations have provided significant time improvements and have enabled breakthrough electronic structure calculations which were not possible before. We are currently experimenting with block techniques that take advantage of the higher performance of SGEMM kernel, and we are considering ways of improving the delivered performance of the SGEMV kernel.