Theor Chim Acta (1987) 71: 105-115

Theoretica Chimica Acta

Strategies for obtaining the maximum performance from current supercomputers

Peter R. Taylor[1]* and Charles W. Bauschlicher, Jr.[2]

[1] ELORET Institute, Sunnyvale, CA 94087, USA
[2] NASA Ames Research Center, Moffett Field, CA 94035, USA

(Received June 27, revised July 6/Accepted July 16, 1986)

The exploitation of special features of the current generation of supercomputers is discussed with particular reference to computational quantum chemistry. Modification of algorithms in the light of the very large memory available on the CRAY 2 is described, and multi-processing of programs is considered for both the CRAY XMP and CRAY 2. The overhead in multi-processing can be reduced to a small percentage even for steps that perform extensive IO. The division of work among CPU's at different program levels is considered for various examples, and optimal levels for the division are discussed.

Key words: Supercomputers — Multi-processing — Vectorization

1. Introduction

The advent of vector processors has been responsible for a very substantial enhancement in the performance of quantum chemistry codes. In many applications, improvements of more than an order of magnitude relative to the fastest scalar machines have been seen on the CRAY 1, CRAY XMP and CDC CYBER 205. However, there seems to be little prospect of another order of magnitude increase in the near future from new single CPU supercomputers: the present machines are close to the limit of what can be achieved in this respect. For large improvements in the performance of quantum chemistry codes it will be necessary

* *Mailing address*: NASA Ames Research Center, Moffett Field, CA 94035, USA

to use multiple CPU's in parallel. As we will show, the simultaneous use of several CPU's in a single job can not only enhance performance, but can also make more efficient use of the large memory and peripheral storage resources of machines such as the CRAY XMP and CRAY 2.

In the present study, we shall describe our experiences in exploiting the resources of the CRAY XMP-48 and CRAY 2 supercomputers installed at NASA-Ames Research Center. While much of the discussion will be devoted to using multiple CPU's we shall also examine the advantages offered by the 256 Mword central memory of the CRAY 2. Our investigations in multi-processing have been restricted to the general architecture of a machine with a few (four in the above cases) very fast CPU's and consequently we shall confine our discussion to multiprocessor supercomputers. Machines featuring a very large number of relatively slow processors - commonly found as polytope configurations such as hypercubes - are outside the scope of the present study.

2. Optimal use of very large memories

We will discuss two examples of computational approaches appropriate for computers with very large central memories. The first example is the generation of full CI wave functions for cases involving 10 000 000 or more determinants; the second is the generation of SCF wave functions for large (more than 250 CGTOs) basis sets without disk storage or repeated calculation of integrals.

The Numerical Aerodynamics Simulation (NAS) project, a national facility operated at NASA-Ames, features a CRAY 2 computer with the maximum configuration of four CPU's and 256 Mwords (268 435 456 64-bit words). With a cycle time of 4.1 ns, the CRAY 2 can reach 420 MFLOPS on a single CPU for matrix multiplication (MXM). However, because of bank conflicts in memory, the performance is no better than 285 MFLOPS per CPU when all CPU's are busy. This is still impressive performance, and it is clear that the machine is ideally suited for a computational scheme which depends heavily on matrix multiplication and which requires large memory resources. The determinantal full CI algorithm of Knowles and Handy [1], based on earlier work of Siegbahn [2] is just such a scheme.

Full details of the determinantal full CI program have been given by Knowles and Handy [1] and will not be repeated here. For CI expansions of more than 1 000 000 determinants, matrix multiplication represents well over 90% of the computational effort. A particular advantage of the large memory of the CRAY 2 is that not only is it possible to handle expansions of over 20 000 000 determinants (28 000 000 is the largest we have employed), it is also possible to decrease the input/output (IO) burden and the amount of disk storage required. Thus, in addition to the usual direct CI memory requirements of a trial coefficient vector $\boldsymbol{c}$ and the corresponding residual vector $\boldsymbol{Hc}$, where $\boldsymbol{H}$ is the Hamiltonian matrix, we also keep the diagonal of $\boldsymbol{H}$ in central memory. This avoids the necessity of reading these elements in each CI iteration, and releases disk space which can be better used to hold earlier $\boldsymbol{c}$ and $\boldsymbol{Hc}$ vectors for the Davidson diagonalization.

Hence, for a calculation involving some 28 000 000 determinants, 84 000 000 words of central memory are required for vectors and the diagonal of $\boldsymbol{H}$; scratch space is also required for setting up temporary arrays [1], and we routinely allocate some 20 000 000 words for this purpose. The full CI code thus uses about 100 000 000 words of central memory. For further details of the full CI calculations, and applications to a number of atomic and molecular systems, the reader is referred to [3–7].

The extensive memory of the CRAY 2 suggests another application in which disk storage is replaced by central memory storage: SCF calculations with the integral list held in memory. In several applications of interest to us, such as the chemistry of small metal clusters, the basis sets used are large (over 250 CGTOs) but the symmetry is relatively high. The total number of distinct, non-zero integrals in many cases is then between fifty and one hundred million, and the overheads of storing and (repeatedly) retrieving such an integral list from disk would be so large that we would normally use the direct SCF method of Almlöf and co-workers [8] to eliminate the IO. Clearly, by constructing the integral list once in memory, we can combine the CPU performance advantage of the conventional approach (that is, calculating the integrals only once) with the advantage of no IO overheads of the direct SCF method. In our preliminary version of such an "in-core" SCF program, the second and later iterations run some 8–10 times faster than the first, during which all the distinct integrals must be generated. It is difficult to compare this time to a direct SCF iteration, since the extensive use of pre-screening of integrals based on density matrix elements in the direct SCF method [8] cannot readily be implemented in the in-core SCF. However, for the worst case for direct SCF, that is, density matrix pre-screening achieves no reduction in the number of integrals to be calculated, it would be possible to perform 17 in-core SCF iterations for the cost of 3 direct SCF iterations, a very useful performance improvement. Even for basis sets of 272 CGTOs, less than 60 000 000 words of memory are required if the point group is of order 12 or higher, while for extended systems almost 600 CGTOs have been treated in 120 000 000 words.

While these examples show the computational advantages to be gained by exploiting large central memories, the schemes used depend on the availability of a large fraction of memory (more than one-third in the full CI case discussed above) to a single user job. This question of the availability of system resources and sharing of resources among user jobs is especially important in the context of multi-processing, as we will show in the next section.

3. Why multi-process?

Multi-processing can be implemented at several levels. The outermost level, and the simplest to implement, is to run a different job on each CPU. No programming effort on the part of users is required: all of the overhead of running multiple processes is borne by the system. However, this approach has several disadvantages. First, there is no speed-up or throughput advantage for each job, and thus the only user benefit is the possibility of running more jobs. The opportunities

for running longer jobs are not necessarily improved. Second, and rather more subtle, is the question of system resource allocation. If a single user job accesses all of the available central memory, all CPU's except the one running the user job will sit idle, as there will be no memory for jobs queued for the other CPU's. This causes considerable inefficiency in throughput, and an obvious solution is to limit jobs to no more than $1/n$ of the central memory, disk storage and solid-state disk (SSD) resources in an n processor system. It should be noted that implementing this policy can actually *reduce* the resources available to users: on a single processor CRAY XMP-14 (that is, one CPU, 4 Mword of central memory) any user job could, in principle, request 4 Mword of memory, while if the system were upgraded to a 4 CPU 8 Mword XMP-48 the above policy would restrict each user to a maximum of 2 Mword. However, there is an obvious extension to this policy of resource allocation: if a user job can make use of several (say, m) CPU's simultaneously, it is reasonable for that user to request m/n of the system resources, since there is no risk of these m CPU's becoming idle.

The possibility of accessing more of a system's resources by simultaneously utilizing several CPU's is an important motivation for multi-processing. Multi-processing itself can increase the throughput of (in effect, speed up) a given job, while very often the opportunity to use more of the system's resources results in further performance improvements. Further, while current operating systems can manage queues of the size required to keep, say, the 4 CPU's of a CRAY XMP-48 busy, scheduling for machines with 16 or more such CPU's would require considerable extension of the operating system capabilities. Again, use of multiple CPU's within a single job would help with system management, since the overall number of jobs required to keep the system busy would be decreased. With these potential advantages of multi-processing in mind, we now turn to the question of how best to implement multi-processing in user jobs.

4. At what level should multi-processing be implemented?

The most basic level at which multi-processing can be implemented is illustrated by dividing the range of a DO loop among multiple CPU's. For example a DO loop running from 1 to 100 000 could easily be broken into 1–50 000 and 50 001–100 000, and each half could run on a different CPU. This is similar in spirit to the vector pipes on the CYBER 205, where the vectors are automatically partitioned across pipes. However, it is more general since the other CPU is free to work on another job when it is no longer required. The disadvantage of this approach is that it requires long loops, as otherwise the overhead of acquiring other CPU's and dividing the work can exceed any potential benefits. Also, much of the time would probably be spent using only one CPU (unless the job was dominated by long DO loops), so that the job should not be allowed more than $1/n$ of the resources.

A higher level of multi-processing would allow sections of code in which parallel independent operations were performed to be partitioned so that several CPU's

could each work on part of the problem. As an example, consider matrix multiplication, $\boldsymbol{A} = \boldsymbol{B} \times \boldsymbol{C}$, using the outer product algorithm (that is, formulated as a sequence of SAXPY [9] (vector = vector + scalar × vector) operations). Each column of the result matrix $\boldsymbol{A}$ is formed independently, and several columns, in arbitrary order, could be generated simultaneously. CRAY Research has implemented this level of multi-processing and has called it micro-tasking. The chief features of micro-tasking are a relatively low overhead associated with generating ("forking") additional tasks and a flexible approach to determining how many additional CPU's to make available to a job - this decision is made by the operating system based on how many processors are currently idle. Clearly, in a busy system few, if any, additional CPU's will be made available, while in an idle or dedicated machine all CPU's may be available. This flexibility in scheduling means that a user code which contains any specific reference to a certain number of tasks or CPU's is unacceptable as a candidate for micro-tasking, at least in the present releast of the software. On the NASA-Ames CRAY XMP-48, a matrix multiplication code micro-tasked along the above lines will run some four times faster than a single CPU code, at least in dedicated mode. In a busy machine we observe a large variation in performance under different loads: this makes it difficult to draw detailed conclusions, or to decide how much of a system's resources should be made available to a micro-tasked job. Micro-tasking is not currently available on the CRAY 2.

Under many circumstances, a code can be divided into different and unrelated tasks. For example, in a direct CI calculation there is no reason why the processing of the various groups of integrals (these are classified according to how many occupied orbital indices appear) cannot occur in parallel on different CPU's. In such a case, the division of work occurs at a high level, and the CPU's are used for a considerable fraction of the total time; scheduling of system resources should therefore account for this. The disadvantage of this level of multi-processing is that in general more extensive program modifications are needed. CRAY Research has also implemented this approach to multi-processing and calls it multi-tasking.

This high level approach to multi-tasking has a particular advantage when some of the scalar overhead associated with different tasks can be divided among CPU's. This is especially important on systems featuring multiple CPU's with high performance vector-processing capability. It is well known that, in a code with an irreducible scalar component, there is a limit to the gain obtainable from vectorization: once the scalar component dominates the timing it is pointless to continue improving the vectorized component. Obviously, multi-processing this vectorized component only exacerbates the problem. However, if the scalar code can also be multi-tasked then a real overall gain from multi-processing is possible. We have multi-tasked (on two CPU's) the MOLECULE Gaussian integral program by duplicating the integral evaluation routines and calling both the original routines and their duplicates in parallel. Virtually all of the calculation is performed using two CPU's and a performance improvement of essentially a factor of two is observed. It is also possible to multi-task much of the overhead in a Slater

integral code, as well as those parts which are highly vectorized, and again a considerable enhancement in performance results [10].

5. Multi-tasking matrix multiplication

Matrix multiplication is one of the key kernels in computational chemistry and it is important to benchmark the multi-tasking of this process. This has been done on both the CRAY XMP-48 and CRAY 2. The most straightforward approach was used: $\boldsymbol{A} = \boldsymbol{B} \times \boldsymbol{C}$ being broken up to generate parts of the result matrix $\boldsymbol{A}$ by making four calls to the system matrix multiply routine, MXM. For matrices of order several hundred, the overhead associated with forking the three extra tasks is very small, and on the CRAY XMP the elapsed time is reduced by essentially a factor of 4 in a dedicated machine - the actual speed-up is 3.99. Since on one processor MXM runs at 200 MFLOPS, the net performance in the four processor version is 796 MFLOPS. On the CRAY 2, the overhead with forking the tasks is higher, but only marginally so. While the CPU of the CRAY 2 is twice as fast as the XMP, not only is the memory four times slower, but the memory latency (bank busy time) is some fifteen times greater. The slower memory does not affect the single-tasked MXM subroutine and, as noted in Sect. 2 above, in a dedicated machine this achieves 420 MFLOPS - twice the performance of the XMP - but in multi-user mode the bank conflicts created by other processes degrade this figure to 285 MFLOPS. When the four processor version of the matrix multiply kernel is run on the CRAY 2, only 1.2 GFLOPS (i.e. 1200 MFLOPS) is achieved, in dedicated or multi-user mode. Bank conflicts, whether from other user processes or from other tasks forked by a given job, thus constrain performance to 4 times the single processor multi-user mode performance. This is less than three times the performance of a dedicated mode single processor version.

Given the large improvement in the MXM performance, the determinantal full CI was multi-processed. The ideal approach would be to divide the code such that each processor formed a temporary matrix of the product of CI coefficients and coupling coefficients, then multiplied this by the integrals, and subsequently by the other coupling coefficients, finally adding the result into the current residual vector $\boldsymbol{Hc}$. Each process could share the CI vector and integrals, but to work properly each would have to access a separate version of $\boldsymbol{Hc}$; these would be merged at the end of an iteration. However, holding four copies of $\boldsymbol{Hc}$ would increase the memory requirements greatly (by 84 million words in our H_2O example given above). With only one copy of $\boldsymbol{Hc}$, code must be added to prevent simultaneous updating of the same elements. Because of this updating complication and the fact that there are only four CPU's on the CRAY 2, the simpler approach of only multi-processing the MXM was made. Since the matrix dimension is quite large and 90% of the time is in MXM, the 4.0 speed up for MXM translates into an overall speed-up of 3.1 in a multi-user environment. (Note that if the timings are made on a dedicated machine the speed-up is less since the one processor version does not suffer from any bank conflicts while the multi-processed version does.) This improvement should extend the capability

of the full CI program, particularly since at present we are limited by the short periods of time for which the system is available to users.

6. Multi-processing the sparse matrix vector product

The sparse matrix vector product is the most time consuming step in the construction of the Fock matrix in an SCF program or in forming the ***Hc*** product in the diagonalization for a conventional CI program, where the list of integrals or Hamiltonian elements, respectively, is explicitly stored on disk. Vectorization of this procedure has been discussed before [11] and in this work we consider multi-processing.

The matrix is real symmetric and 0–10% dense, and stored on disk by rows with the column index of each element packed into the low order bits. The example is taken from [11]. In order to simplify the notation we consider the example of diagonalizing a matrix of dimension, MDIM, using the simultaneous vector approach of Liu [12, 13]. On each iteration of the product of the matrix and each of the NROOT vectors is needed. The scalar code is

```
C sparse matrix version of symmetric matrix-vector code
      DO 1 I=1,MDIM
C The matrix is stored by rows,
C NZ is the number of non-zeros in the row
C MASK is an integer of the form (2**n)-1 where MASK>MDIM.
C For our applications we usually choose n=16.
      READ(IU)NZ,(BUF(J),J=1,NZ)
      DO 2 K=1,NROOT
      DO 3 JX=1,NZ
      J=AND(BUF(JX),MASK)
      HC(J,K)=HC(J,K)+C(I,K)*BUF(JX)
      HC(I,K)=HC(I,K)+C(J,K)*BUF(JX)
3     CONTINUE
2     CONTINUE
1     CONTINUE
```

It is possible to vectorize this code on the CRAY's by the use of SPDOT and SPAXPY, and the code is;

```
C FORTRAN version illustrating SPDOT and SPAXPY
      DO 1 I=1,MDIM
      READ(IU) NZ,(BUF(J),J=1,NZ)
C Unpack labels into scratch array ISC
      DO 10 J=1,NZ
10    ISC(J)=AND(BUF(J),MASK)
      DO 2 K=1,NROOT
C Gather together elements of C SPECIFIED BY THE INDEX ISC
      DO 20 J=1,NZ
```

```
20   SCR(J) = C(ISC(J),K)
C Form dot product
     SUM = 0.
     DO 21 J = 1,NZ
21   SUM = SUM + SCR(J)*BUF(J)
     HC(I,K) = HC(I,K) + SUM
C SPDOT can replace loops 20 and 21.
C    HC(I,K) = HC(I,K) + SPDOT(NZ,C(1,K),ISC,BUF)
C GATHER TOGETHER THE ELEMENTS OF HC SPECIFIED BY THE
INDEX ISC
     DO 22 J = 1,NZ
22   SCR(J) = HC(ISC(J),K))
C NOW PERFORM SAXPY (SUM A*X + Y) OPERATION
     DO 23 J = 1,NZ
23   SCR(J) = SCR(J) + C(I,K)*BUF(J)
C SCATTER MODIFIED ELEMENTS BACK INTO HC
     DO 24 J = 1,NZ
24   HC(ISC(J),K) = SCR(J)
C SPAXPY can replace loops 22, 23 and 24.
C    CALL SPAXPY(NZ,C(I,K),BUF,HC(1,K),ISC)
2    CONTINUE
1    CONTINUE
```

We have considered multi-processing both of these codes on the CRAY XMP-48 using all four processors. Unlike the full CI, where it was not possible to have multiple copies of HC, for both the conventional diagonalization and the SCF calculation the matrix dimension is sufficiently small that this presents no problems. Therefore we have one copy of C (the density matrix in the SCF) and four copies of HC (the Fock matrix in the SCF). The example uses IO which reads only one row at a time, this is for illustrative purposes only. In practice the IO is performed in long fixed blocks; each block contains a series of logical records which start with a word specifying the row and the number of non-zeros. If a row spans a block, both parts of the row are started by a specifier word. This is critical to ensure that each block in the file can be processed totally independently. The program organization is to form the initial C, then to fork four tasks, each of which looks like the sample code above. Since all tasks read the same file, if the machine is busy the first task will process the entire file, whereas if other processors are available they will each perform part of the construction. The program waits until all tasks are finished and merges the four results. From the result a new trial function is made. Currently this is done on one processor and the memory could be reduced to that of the single-tasked version. Memory is an important characteristic of a job and the one processor version uses $2 * n$ words, where n is the dimension of the matrix. The four processor version obviously needs $3 * n$ new arrays to hold their contribution to HC. Thus the average memory per processor is *decreased* in the multi-processor version; this is important in the overall scheduling of the system since it effectively leaves more memory available

for other jobs. It is obviously important for system efficiency that a job which performs extensive IO should leave memory for other jobs.

For a 3% dense matrix of dimension 20 000 the vector code is about 8 times faster than the scalar code on the CRAY XMP. When the problem is multi-processed, the total CPU time increases because of the overhead in forking tasks: this overhead is some 5% of the time for the scalar code, but 40% for the vector code. This illustrates well the need to multi-task at the highest possible level, as discussed in Sect. 4. In fact, in a busy machine, one CPU completes the entire problem, in the vectorized version, before the other tasks can be executed. Clearly, multi-processing must be implemented at a sufficiently high level to ensure that tasks do not become too small and that the balance of work is shared properly between the tasks.

It is interesting to note that a larger improvement is obtained by vectorization (a factor of 7.9) than by multi-processing the scalar code among four CPU's (a factor of 3.8). For the more time-consuming scalar case, the 3.8 improvement is quite acceptable and worth the effort of coding. For a case which is 10 times larger, even the overhead in the vector version would be quite low and multi-processing would be worthwhile. However, this example has only considered CPU time – as noted above, the IO wait time can present a problem. If the matrix is stored on disk, the disk IO rate cannot keep up with even one processor, hence increasing the number of processors cannot reduce elapsed time. In a multi-user environment this effect is very hard to measure: we have observed timing for a one processor run with IO wait times far longer than four processor runs made on other occasions. Regardless of these individual runs, overall the four processor version does not obtain the same elapsed time speed-up as CPU speed-up, and thus multi-processing does not fully solve the problem. This IO wait time problem is even more severe for the vectorized version of the code, of course, where the time to process the block is much smaller. There are two ways to solve this problem. On the CRAY 2 the matrix could be held in core for many cases. Also, since all four processors can effectively work on the problem all the memory could be made available to the job and a large problem could be treated. On the XMP-48 this is not an option, but storing the matrix on the SSD is. With a 20 Gbit/second transfer rate, as little as four floating point operations per word transferred are needed to make CPU power, not IO, the limiting factor. Therefore, once the matrix is stored on the SSD the IO wait time is effectively eliminated (provided at least four products are to be formed), and the full power of the four processors can be applied to this problem.

An obvious alternative approach to the IO problem is to divide the data into several files. If these are allocated to independent disk units a substantial enhancement in IO performance will be obtained. The NASA-Ames computer center is also investigating "striping' data across several packs to reduce IO wait time on sequential IO. Such alternatives need to be investigated: the current SSD is limited to 128 Mwords, and no doubt when multi-processing becomes more widely used more scratch space will be required.

7. Conclusions

The very large memory of the CRAY 2 can be used to treat special approaches like the full CI which represent very important calibration points for all methods. The memory can be used to store data that would be held on disk on most machines. This eliminates the IO wait time associated with disk IO. Once the disk IO wait time is eliminated, the idea of using multiple CPU's to process this memory-resident data becomes a real option.

The multi-processing of matrix multiplication is shown to achieve 796 MFLOPS on the XMP-48 and 1.2 GFLOPS on the CRAY 2. Since large portions of our codes are limited by the speed of this operation this translates to a useful speed-up in the computational chemistry codes. Of course, as the number of processors is increased it is not sufficient to concentrate only on the MXM step. This problem of the scalar setup overhead is a problem in all highly vectorized codes and multi-processing can amplify it.

It is shown that a step which is in principle IO bound can be multi-tasked on the XMP-48 by moving the data onto the SSD. Unlike the CRAY 2, it is not possible to move the data into central memory. When the job is run in a multi-user environment, it is possible that such a multi-tasked problem could still work well, provided it used little memory and the other jobs did little IO. However, it does appear that an SSD is required to achieve high performance for problems in which only a few floating point operations are performed on each piece of data read in. We are currently examining the case of problems in which N operations are performed on each of N^2 data items read from disk.

We have described our first attempts to utilize fully the memory of the CRAY 2 and the multi-processing capabilities of the CRAY 2 and CRAY XMP-48. Significant enhancements in the performance of computational chemistry codes can be achieved by using multi-processing and by taking maximum advantage of large memories. Extension to architectures featuring more than about 10 processors will probably require more extensive alteration of our current approaches.

Acknowledgements. The authors would like to thank ZeroOne Systems and the NAS project for help in using multi-tasking and for making dedicated time available so that reliable timing information could be obtained. Helpful discussions with D. Calahan and H. Partridge are gratefully acknowledged; S. R. Langhoff made many valuable comments on a preliminary version of the manuscript.

References

1. Knowles PJ, Handy NC (1984) Chem Phys Lett 111:315
2. Siegbahn PEM (1984) Chem Phys Lett 109:417
3. Bauschlicher CW, Langhoff SR, Taylor PR, Partridge H (1986) Chem Phys Lett 126:436
4. Bauschlicher CW, Langhoff SR, Taylor PR, Handy NC, Knowles PJ J Chem Phys, in press
5. Bauschlicher CW, Langhoff SR, Partridge H, Taylor PR J Chem Phys, in press
6. Bauschlicher CW, Taylor PR J Chem Phys, in press
7. Bauschlicher CW, Taylor PR J Chem Phys, Submitted for publication
8. Almlöf J, Faegri K, Korsell K (1982) J Comput Chem 3:385

9. Lawson C, Hanson R, Kincaid D, Krogh F (1979) ACM Trans Math Software, 5:308
10. Partridge H: unpublished work
11. Bauschlicher CW, Partridge HJ: Comput Chem, submitted for publication
12. Liu B (1979) In: Proceedings of the 1st West Coast Theoretical Chemistry Conference. IBM San Jose (1979)
13. Davidson ER (1975) J Comput Phys 17: 87