Optimistic Parallel Computation: An Example from Computational Chemistry

Emily Angerer Crawford
emily@cc.gatech.edu
School of Elec. and Comp. Engineering

Karsten Schwan
schwan@cc.gatech.edu
College of Computing

Sudhakar Yalamanchili
sudha@ee.gatech.edu
School of Elec. and Comp. Engineering
Georgia Institute of Technology, Atlanta, GA 30332

Abstract

Performance penalties due to synchronization are a common concern in parallel programming. This paper describes a technique for avoiding such penalties when they occur in shared data structures. This technique may be used with a segment of code in which two updates are required for any element of a shared array and the values stored in the array are known *a priori*. Traditional approaches enforce the correct ordering of write operations using locks, but this can be time-consuming and drastically reduce the benefits of using a parallel machine. Instead, we propose using an optimistic approach where the solution is calculated without any locks, during which statistics on memory writes are kept. These statistics can then be used to determine whether the calculation resulted in a correct answer or if the answer should be recalculated due to incorrect ordering of write operations. This scheme is tested on an implementation of the Møller-Plesset perturbation theory energy calculation for closed-shell molecules and significant speedups are shown, as demonstrated on an SGI PowerChallenge.

1 Introduction

Shared memory parallel machines are often chosen as target platforms for porting existing applications due to the relative simplicity of the programming model. However, in developing an application on such a system it is necessary to find formulations of the solutions that admit to sufficient parallelism and are not overshadowed by synchronization and communication requirements. In accessing shared data structures we are primarily concerned with ensuring that data dependencies are honored thereby avoiding data hazards (e.g., read-after-write). This is usually achieved via the use of mutual exclusion locks at a considerable expense in execution time performance. The resulting overhead often precludes the application of parallel processing techniques to improve the performance of computationally intensive applications. However, in certain applications data races may occur infrequently in practice, while being relatively easy to detect after they have occurred. In such applications, parallel computation may proceed optimistically assuming that hazards will not occur. If the occurrence of a hazard is detected, the result may then be recomputed, possibly guaranteeing correctness via the use of locks. When the cost of hazard detection and the frequency of occurrence are sufficiently low, substantial performance improvements over serial execution become possible. This paper presents an evaluation of an implementation of such an approach for an application from the domain of computational chemistry.

The computational chemistry application that we have parallelized in this research is the calculation of the second-order Møller-Plesset perturbation theory energy for closed-shell molecules. The majority of the computational effort in the serial implementation can be found in two components: *trans* and *mbpt*. The component *trans* utilizes a very large array that is accessed with no predictable pattern. Each element of the array is updated twice with addition operations: first to an initial value of zero, and second to a non-zero value. If elements of the array are locked to enforce correct sequences of write operations, the overhead of the locks becomes prohibitive. However, if the entire array is locked for each write, parallelism is inhibited. If no synchronization is enforced, two processors may try to update a value simultaneously, resulting in an incorrect value. By maintaining a count of the number of additions to zero values and the number of additions to non-zero values it can be determined whether a hazard actually occurred. In practice such hazards rarely occur. Since the calculation is highly parallel, re-running this portion of the application when an error is detected is relatively inexpensive.

Examples of optimistic computation can be found in many areas of computing. In parallel discrete event simulation, events can be executed optimistically as they are received. Corrective action is taken when a violation of causality constraints between event execution is detected. In the implementation of direct mapped cache memories, entries are assumed to be available in the cache and are fetched in parallel with the access checks performed on the cache directory. The cache access is prevented from completing and a cache miss is generated if the access checks indicate that the required cache line is actually not present [HB84]. A simple strategy for branch prediction in pipelined processors is to assume that branches are not taken (or taken) and to continue fetching instructions. If the branch decision is subsequently found to be in error, the instructions in the pipeline are flushed and prevented from completing. All of these examples lead to significant improvements in computing system performance be exploiting the behavior of application programs. This paper describes the successful application of an optimistic approach to the synchronization of a shared memory implementation of an important application in computational chemistry.

The following section describes the structure of the application and the nature of the computation. Section 3 discusses our parallelization approach and section 4 evaluates the success of our algorithms. The paper concludes with a final description of the future directions of this work.

2 An Example from Computational Chemistry

2.1 Overview: The Second-Order Møller-Plesset Perturbation Theory Energy Calculation for Closed-Shell Molecules

In the quantum mechanical study of a molecule, it is generally necessary to begin with the Hartree-Fock self-consistent-field (SCF) method [Roo51]. However, the SCF approach fails to consider the correlation of electronic motion. Thus, one of the first steps beyond SCF is the calculation of the second-order Møller-Plesset perturbation theory energy (MP2) [MP34], which does include electronic correlation. However, it is well-known in the field of electronic structure theory that the calculation of this MP2 correlation energy can be extremely computationally demanding, requiring significant CPU time and memory. This motivates the search for parallel formulation.

Parallelization of this application is relatively straightforward and can be achieved via domain decomposition. However, exploiting this parallelism is a significant challenge due to heavy demands on memory resources and unpredictable memory reference patterns. Due to these irregular access patterns, a shared memory solution appears preferable, although message passing implementations have been studied [MD95].

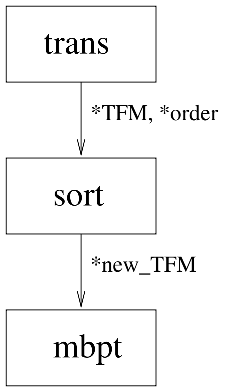

Figure 1: Structure of the program before parallelization, including the sort between *trans* and *mbpt*.

There have been some previous attempts to study the parallelization of similar electronic structure theory calculations. Watts and Dupuis examined the same application as part of the HONDO quantum chemistry program in 1988 [WD88] using the Loosely Coupled Array of Processors (LCAP) systems [Cle85, CCD+84] at IBM in the mid-1980s. Their implementation also used shared memory and they reported a modest speedup; however, the implementation was only useful for the native system and was limited in its ability to be scaled to larger machines. Further, the LCAP system never evolved beyond the experimental stages, so the parallel application was of limited use to the general scientific community. In a recent paper, Márquez and Dupuis [MD95] published the current efforts with MP2 in the HONDO program, which includes an implementation with the Parallel Virtual Machine (PVM) [GBD+94] programming environment. They parallelized a greater portion of the MP2 problem than presented here, but did not eliminate the communications bottleneck generated by the domain decomposition. Their overall speedups are excellent, but on the more data-oriented portions of the computation the distributed memory platform encounters significant performance penalties due to inter-processor communications. As a result, these portions of the computation do not realize as significant an improvement as some of the others. In contrast, our approach has been to avoid the excessive communication overhead resulting from domain decomposition by creating a shared memory implementation and utilizing an optimistic approach to synchronizing access to shared data structures.

2.2 Mathematical Formulation

The Møller-Plesset energy calculation computes the second order perturbation theory energy for a closed-shell molecule. The energy can be written as:

$$E_{MP2} = \frac{1}{4}\sum_{ij}^{occ}\sum_{ab}^{vir}\frac{(ia|jb)[2(ia|jb) - (ib|ja)]}{(\epsilon_i + \epsilon_j - \epsilon_a - \epsilon_b)} \tag{1}$$

where i and j are doubly-occupied molecular orbitals and a and b are unoccupied molecular orbitals for a given molecule. (There are no singly-occupied orbitals for this calculation, as we are working with only closed-shell molecules.) The notation $(ia|jb)$ represents a two-electron integral over spatial orbitals, in Mulliken notation, for a particular i, j, a, and b, and ϵ_k is the energy of orbital k[SO89], [Wil87].

2.3 Structural Description of the Computation

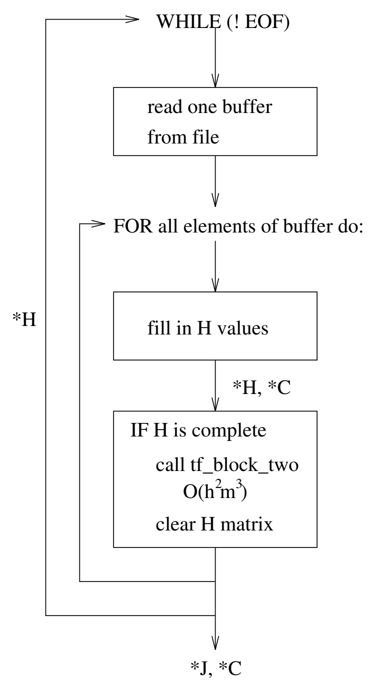

Figure 2: Structure of *rdtwo* before parallelization.

The computation proceeds in two primary stages (see Figure 1). The first is a transformation function (*trans*) which sets up the two-electron integral matrix for use in the second stage by transforming the two-electron integrals from the atomic orbital (AO) basis to the molecular orbital (MO) basis. The function *trans* itself is decomposed into two main blocks, *rdtwo* and *trans_two*.

The function *rdtwo* reads in a large file containing the AO basis functions and performs a pre-transformation. The initial structure of *rdtwo* is as shown in Figure 2. Each iteration performs a computation of time complexity $O(h^2m^3)$, where h is the number of irreducible representations and m is the number of orbitals per irreducible representation [Cot90]. h and m are generally small numbers; h is a maximum of 8, and m is on the order of 10^1 to 10^2, with a maximum of approximately 200. The size of the input file varies according to the number of basis functions; the problems we dealt with here utilized under 1000 buffers of data (where a buffer is 2980 doubles, read in as a block), but much larger cases are typical. The function *trans_two* performs the actual transformation, beginning from the intermediate matrices produced by *rdtwo* and producing a complete two-electron integral matrix. The function *trans* is of time complexity $O(h^4m^5)$ at its deepest point, where h and m have typical values described above.

The energy calculation is performed in the second stage (*mbpt*, see Equation 1). The function *trans* produces values for the numerator and the orbital energies are provided for the denominator. In the original application the output of *trans* was sorted prior to computation in *mbpt*, but this proved inefficient in the parallel application, as the added expense of an indirect memory access per element was less than the total time required for the sort. In a parallel implementation, much of the expense is eliminated by the parallelization, so the effort involved in eliminating the sort operation was more beneficial than it would have been in the serial application. The function *mbpt* is of time complexity $O(N^2n^2)$, where N is the number of doubly-occupied MO's and n is the number of unoccupied MO's.

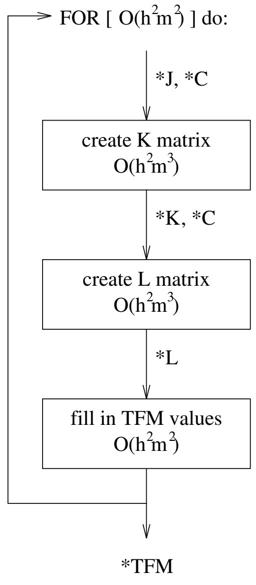

Figure 3: Structure of *trans_two* before parallelization.

3 Parallelization Approach: Optimistic Computation

The functions *trans* and *mbpt* are implemented using a multiprocessor threads package, Cthreads [MEG94], which supports thread-based parallelism and provides portability between a variety of shared-memory multiprocessor and uniprocessor platforms. Our implementation platform was an SGI PowerChallenge 12-processor shared memory machine. The initial code was based on the Psi 2.0 suite of computational quantum chemistry programs[JSS$^+$94]. Both *trans* and *mbpt* are structured such that parallelization can be achieved via data decomposition.

3.1 Parallelization of *trans*

The basic structure of *trans* before parallelization is based on the Saunders-van Lenthe algorithm [SvL83]. The function *trans* is composed of two segments, input processing (*rdtwo*) and the two-electron integral transformation (*trans_two*). The structure of *trans_two* is illustrated in Figure 3. At the core of *trans_two* is a series of nested loops that modify the elements of a large four-dimensional matrix, TFM. The computation produces multiple updates to each matrix element, i.e. each element accumulates the sum of values computed over more than one iteration.

However, due to the nature of the computation each matrix element is updated exactly twice and all matrix elements are initialized to zero. The element update appears as shown below:

```
if (ij == kl)
   TFM[ijkl] += 2.0 * (L[lt][kt] + L[kt][lt]);
else
   TFM[ijkl] += (L[lt][kt] + L[kt][lt]);
```

Parallelization of this computation was relatively straightforward (see Figure 4). The program spawns multiple, concurrently executing threads, including one master thread and multiple slave threads. Each thread runs on a distinct processor and executes a set of outer loop iterations. The number of iterations per thread is chosen to balance the granularity of the computation. Since we are working with shared

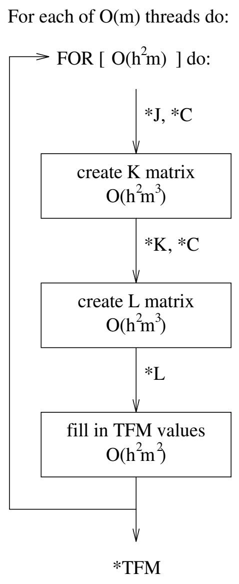

Figure 4: Structure of *trans_two* after parallelization.

memory, we allocate TFM to reside in shared memory along with any common input data. However, any other data that is written within the inner loop of *trans*, including most intermediate arrays, is made local to each thread.

While TFM is globally shared, individual processors may cache the matrix elements. We can ensure that all updates occur in order on the main memory copy of the elements by serializing updates to the complete matrix. This approach clearly precludes any performance via parallel computation of matrix values. However, we can exploit application characteristics to realize a more efficient and highly parallel implementation.

3.1.1 A Simple Example of Optimistic Execution

The following simple example illustrates our approach. Suppose that there are two threads updating a single array A in shared memory. All elements of A are initially zero, and exactly two new values will be added to each value in A. Suppose that A is five elements long, and that the updates occur in each thread in the order shown in case 1 of Table 1.

Each thread reads the values in A at the beginning of a given cycle, and updates and writes a value to A at the end of a cycle. When thread 1 caches elements 1 (the first time), 4, and 0, the values will be 0 prior to update. However, elements 1 (the second time) and 3 are non-zero before being updated. We will call the first case a “zero-write” and the second a “non-zero-write”. Thread 1 records the number of zero-writes (3) and non-zero-writes (2). Thread 2 will find as it caches its values that elements 2, 4, and 3 are zero. Thus three zero-writes and two non-zero-writes are recorded for thread 2. As each thread completes, its local counters are summed in the master thread, resulting in 6 zero-writes and 4 non-zero-writes. Since each element should have been updated twice (once as a zero-write and once as a non-zero-write) the two counts should be equal. Inequality denotes the occurrence of a hazard and incorrect computation. We can see from the table that the both threads attempted to update $A[4]$ simultaneously. When this occurs in parallel execution, only one thread's operation will be successful, which is why the results are deemed *unsafe*. It is not possible to determine which element value is incorrect, only that *some* element value is

	Case 1			Case2		
Time	Thread 1	Thread 2	A	Thread 1	Thread 2	A
init			0 0 0 0 0			0 0 0 0 0
0	A[1]	A[2]	0 x x 0 0	A[1]	A[2]	0 x x 0 0
1		A[2]	0 x x 0 0		A[2]	0 x x 0 0
2	A[4]	A[4]	0 x x 0 x	A[4]		0 x x 0 x
3	A[0]	A[3]	x x x x x	A[0]	A[4]	x x x 0 x
4	A[1]		x x x x x	A[1]	A[3]	x x x x x
5		A[0]	x x x x x		A[0]	x x x x x
6	A[3]		x x x x x	A[3]		x x x x x

Table 1: An example of memory accesses for two threads on an array A. "x" denotes a non-zero value stored in the array, indicating at least one write update has occurred.

incorrect. This is why the matrix must be recomputed.

Now suppose that execution proceeds as in case 2. In this case, thread 1 will detect zero-writes on elements 1, 4, and 0 and non-zero-writes on elements 1 and 3. Thread 2 will detect zero-writes on elements 2 and 3 and non-zero-writes on elements 2, 4, and 0. The main thread will collect final counts of 5 zero-writes and 5 non-zero-writes, which indicates that every element had exactly one zero-write and one non-zero-write. At that point it can be asserted that the matrix has been correctly updated and that the computation can proceed.

It would be impossible for the count of non-zero-writes to be higher than that of zero-writes. Furthermore, the difference in the counter values must be some multiple of 2: a missing update decrements one counter and increments the other.

3.1.2 Extension of the Approach

This example can be easily extended to much larger matrices and many more threads as long as each element of the matrix at issue is updated exactly twice (or not at all, in which case the element does not contribute to zero-writes or non-zero-writes). We characterize this approach as an "optimistic" execution, as we implicitly assume that there will be no hazards and perform hazard dectection on completion. By having eliminated what would have been necessary locks on either the entire array or on individual elements of the array, we have significantly sped up computation. In practice, we have found that the occurrence of a hazard necessitating re-computation is quite rare. Detailed performance results are presented in section 4.1.

A special case arises if any of the values we are adding to an array element is 0. Suppose in case 2 that thread 1 adds the value 0 to $A[1]$ on the first zero-write. The second update will then be incorrectly recorded as a zero-write. It is incorrect since zero-writes are intended to record the first update of an array element and non-zero-writes record the second update. The simple solution to this case is to *decrement* the zero-write counter. The second update will then be recorded as a zero-write and both counters must still be equal for hazard free operation. If a thread is adding a zero value to a non-zero entry, no special accounting needs to be performed; it can be recorded as a non-zero-write element.

Finally, we have the case of both updates to an element being zero. This case can be handled by updating the element with a special notation, e.g., negative numbers if all matrix elements are positive values or extremely large values if all matrix elements are small values. A second update with a zero can be recorded as a non-zero update and the original approach will be correct. (If two threads try to write the out-of-range value to the same location at the same time, it will show up as a potential error in the final accounting, so no check needs to be performed on the entire array to reset the value to a true 0.)

We can also generalize the optimistic approach to fit any case where the values stored in the array are known *a priori*. In this case, two copies of the array must be maintained. Instead of comparing the stored value to 0 before an update, one can compare it to the *a priori* value, and, if the values match, record that operation as a zero-write. In this case, we could still use special values in multiple 0 value updates, and the non-zero-write would need to restore the value from the *a priori* values array before performing the update operation. However, this more general situation would be extremely costly in terms of memory (in our largest test case, we would require an additional 176 MB of memory), although performance should not otherwise be affected.

This approach cannot be generalized to fit any case where the values stored in the final array are updated more than twice. For example, consider the case where exactly three writes are performed on any element. Even though we would expect the final count of non-zero-writes to be exactly twice the final count of zero-writes, this would only detect conflicts between a zero-write and a non-zero-write. If two non-zero-writes were attempted at the same time, both would still be accounted for in the non-zero count, and no error would be detected.

3.1.3 Application of the Optimistic Approach to *trans_two*

We are able to apply the optimistic method to *trans_two*. TFM is the array in question, initially zero in every element. This particular calculation was designed so that no element would have a final value of 0, i.e., the original designers of the application were able to avoid expending CPU time calculating a value of 0. Thus we never encounter the situation where two 0 values are written to a single location, so we are able to use the original method of handling 0 values being added in zero-write cases.

The final segment of code for calculating TFM values is as follows:

```
if (ij == kl) {
   gd->TFM[ijkl] += 2.0 * (L[lt][kt] + L[kt][lt]);
}
else {
   if (gd->TFM[ijkl] == 0) {
      tmp = L[lt][kt] + L[kt][lt];
      if (tmp == 0) /* remove self and other half from accounting */
         local_zeroes--;
      else {
         gd->TFM[ijkl] = tmp;
         local_zeroes++;
      }
   }
   else {
      local_nonzeroes++;
      gd->TFM[ijkl] += (L[lt][kt] + L[kt][lt]);
   }
}
```

Separate counters for zero-writes and non-zero-writes are maintained in each thread, and the final counts are added to a global counter as each thread completes execution.

3.2 Parallelization of *mbpt*

mbpt was structured as follows:

```
FOR each doubly-occupied orbital i
   FOR each unoccupied orbital a
      calculate ia position
      FOR each doubly-occupied orbital j
         calculate ja position
         FOR each unoccupied orbital b
            calculate ib, jb positions
            calculate iajb, ibja positions
            EMP2 += TFM[iajb] * (2 * TFM[iajb] - TFM[ibja])
                    / (e[i] + e[j] - e[a] - e[b])
         END FOR
      END FOR
   END FOR
END FOR
```

Parallelization of *mbpt* was straightforward. TFM, the eigenvalue array e, and an array $EMP2$ (to store partially summed $MP2$ values) are shared data. A number of threads are needed to process outer loop iterations in parallel, each writing to only one element of $EMP2$ and avoiding the use of locks. Each thread signals termination to a main thread, where the final summation is computed.

The current implementation uses on-the-fly lookup as a more efficient alternative to pre-sorting prior to *mbpt*. Therefore in the above code, references to i, j, a, and b inside the loops are to be considered the lookup of those values.

4 Performance Evaluation

We tested our application on an SGI Power Challenge with twelve 75 MHz R8000 processors, 2 GB of RAM, and 3.5 GB of disk space. The SGI PowerChallenge is a shared-memory, bus-based architecture machine. Each node has a 16 KB data cache and a 16 KB instruction cache as well as a 4MB secondary data/instruction cache. Memory is 8-way interleaved for improved access times. The R8000 is a 64-bit architecture; however, this application was run in 32-bit mode because of the inherent design of the PSI libraries.

Some additional testing was performed on a 64 processor KSR-2, which is a NUMA (non-uniform memory accesss) distributed shared memory, cache-only architecture with an interconnection network that consists of hierarchically interconnected rings, each of which can support up to 32 nodes. Each node includes 32 MB of main memory used as a local cache and a higher performance 0.5 MB sub-cache. CPU clock speed is 40 MHz. The results of this testing are not explicitly presented here, but we refer to some of the more interesting points in order to observe the scalability of the algorithm and the effects of different memory systems on this application.

Our testing was performed using various basis sets for water. The water problem can range from fairly small (under a dozen basis functions) to quite large (hundreds of basis functions), and solutions are well-known. The size of the basis set determines both the size of the input files and the approximate execution time.

In this study, we used five basis sets for timing the sequential and parallel versions. These are illustrated in Table 2: i) a triple-zeta plus two sets of polarization functions on all atoms, TZ2P [Huz65, Dun71], ii) a TZ2P plus a single set of f-type functions on the oxygen atom and a single set of d-type functions on the hydrogen atoms, TZ2P+f [Huz65, Dun71], iii) a correlation-consistent polarized-valence double-zeta basis set, cc-pVDZ on the hydrogen atoms with a cc-pV quadruple-zeta basis set on the oxygen, cc-pVQZ, which we call cc-pV1 [Dun89], iv) a cc-pV triple-zeta basis set, cc-pVTZ on the hydrogen atoms with a cc-pVQZ basis set on the oxygen, which we call cc-pV2 [Dun89], and v) a cc-pV 5-zeta basis set, cc-pV5Z on the oxygen atom with a double-zeta plus a single set of d-type functions on the hydrogen atoms, which

Basis Set	Number of Functions	Size of TFM
TZ2P	44	490,545
TZ2P+f	66	2,445,366
cc-pV1	80	5,250,420
cc-pV2	100	12,753,775
cc-pV3	115	22,247,785

Table 2: Characteristics of the test cases.

	TZ2P	TZ2P+f	cc-pV1	cc-pV2	cc-pV3
Sequential	0.1250	0.623	1.324	3.095	5.181
Parallel	0.0964	0.468	1.043	2.444	4.291

Table 3: Memory allocation and initialization times in seconds for *trans_two*.

we call cc-pV3 [Dun89]. These basis sets were chosen strictly for their representative sizes.

The most significant memory demand came from TFM, as described above. The size, in terms of double precision floating point numbers (8 bytes), of the array for each basis set is shown in Table 2.

4.1 Performance of the Optimistic Algorithm

The overall speedup of *trans_two* is limited by the setup time required, the number of iterations (i.e., repetition of the core calculations due to detected errors) necessary for hazard-free computation, and the speedup of a single iteration. Although it required no repetitions, the memory demands of the sequential version were so significant that the setup time would have been comparable to that of the parallel version. In practice, sequential setup time is generally higher than parallel setup time, as shown in Table 3, although setup time was relatively insignificant in all cases.

For a fixed problem, the number of updates is fixed. As the number of processors increases, the probability of simultaneous updates to the same element by two processors increases, i.e., the probability of a hazard increases. We did not see a rise in the number of hazards as the problem size itself increased; this result is not unreasonable, as we are increasing the total number of writes while keeping the number of threads constant. In the vast majority of cases, only a single iteration was required; in the worst case, which occurred quite rarely, four iterations were needed. The average number of iterations required for a given number of processors and a given basis set was less than or equal to 2.

Performance of a single repetition was extremely good, as shown in Figure 5. Speedup was close to linear in all cases. It is expected that the linearity would continue until the number of useful processors (equal to the maximum number of threads as allowed by the algorithm) had been exceeded, at which point we would see a natural decline in performance.

4.2 Scalability of Optimistic Execution

Performance of the work portion of *trans_two* remained quite good for smaller numbers of processors, and an excellent speedup was possible with every basis set. However, the increasing number of repetitions required for the larger processor sets, as shown in Table 4, degraded the speedup (see Figure 6). However,

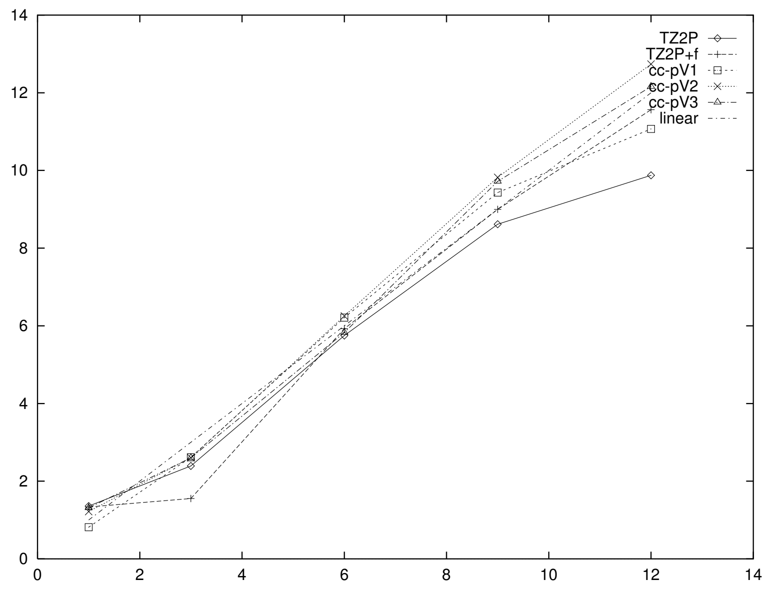

Figure 5: Speedup of a single repetition vs. number of processors in *trans_two*.

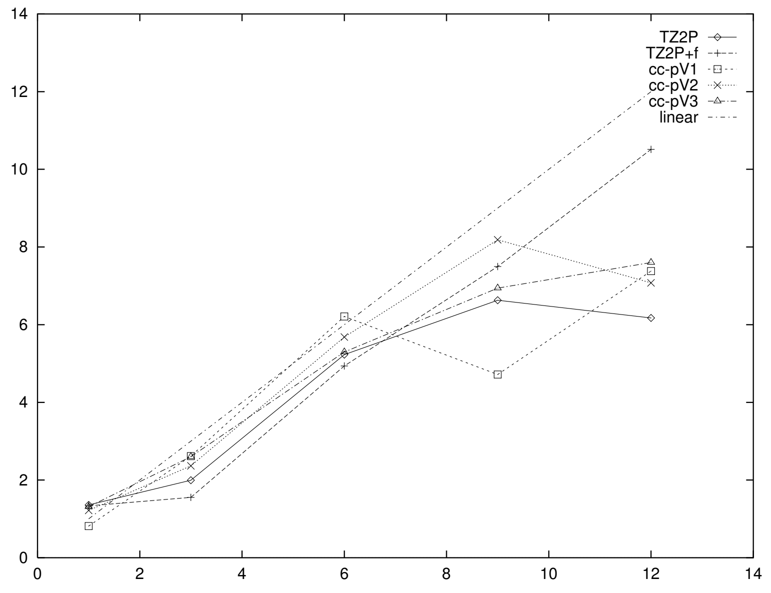

Figure 6: Speedup of *trans_two* work vs. number of processors.

Number of Processors	Avg. passes through code	Probability of a hazard	Max. passes through code
1	1.00	0.00	1
3	1.06	0.06	2
6	1.10	0.10	2
9	1.42	0.26	4
12	1.52	0.40	4

Table 4: Probabilities and costs of the optimistic approach for increasing processor set sizes.

total run time continued to decrease as the number of processors increased, with a few exceptions. This shows that we are able to use a large number of processors effectively, although peak utilization is reached on smaller processor sets. However, if the number of processors were increased beyond the 12 available to us on the SGI we would expect to see more repetitions necessary on average and performance could become quite poor.

Our tests on the KSR-2 confirm this; as the number of processors increased, we saw that considerably more repetitions of *trans_two* were necessary. The speedups remained good for larger processor sets, since our speedups were superlinear to begin with, but peak speedup was seen at 10 or 15 processors. This shows that we are incapable of utilizing a larger number of processors even though we are able to create many threads, each doing a relatively small amount of work. If we have fewer threads, each doing more work but with less likelihood of a memory hazard, we waste less processing time overall. Our results indicate that the optimistic approach is very useful, but a more coarse-grained approach to a problem will yield better results. A fine-grained approach results in more failures and diminishing returns.

It is useful to note that peak performance on a small processor set does not limit the applicability of this approach. It is anticipated that typical workstations within the next decade will have multiple processors available for use by a single programmer. It is very likely that those who program scientific applications will be using this type of machine and not a massively parallel system for their applications, so to see excellent performance only up to six or nine processors would still be of use for applications in the genre we are recommending.

4.3 Performance of *rdtwo*

Timing results for *rdtwo* and speedups for the same are plotted in Figure 7 and Figure 8. We chose only to run this portion of the code for the four smallest basis sets; in larger cases the memory requirements of the input file became prohibitive. Parallelization of the I/O segment of *rdtwo* and integration of the I/O segment with the work segment (as in the sequential version) would overcome this difficulty; however, this was beyond the scope of our current work.

As is apparent from the graphs, our algorithm was very successful for parallelizing the work portion of *rdtwo*. We observed slightly sub-linear speedups for all processor set sizes and basis sets. It appears that if the number of processors were increased, the larger cases would continue to show a substantial speedup.

4.4 Effects of the Memory Model on *mbpt* Performance

As *mbpt* consists almost entirely of memory accesses and requires only minimal computation, the memory model of the underlying architecture substantially affects performance of this segment of the program. The SGI is a uniform memory access (UMA) shared memory machine. Because of this, data remains equidistant to all nodes, so the access times for elements of TFM are consistent regardless of where a given element was calculated initially. Memory access is extremely efficient on the SGI, so the sequential version of *mbpt* was not very time-consuming (less than 1 s) in any of our test cases. Therefore parallel results of this segment of code were not particularly interesting. However, in all but the two smallest cases we see a decline in total run time for *mbpt* as the number of processors increased, as shown in Figure 9, and the speedup over the sequential case was slightly greater than 1 in most cases.

The KSR-2, however, is a NUMA machine, and whether a memory element is local or non-local can significantly affect access times of data. When the test cases became large enough that all of TFM could not be stored on a single node, then the sequential case became very slow because of the number of non-

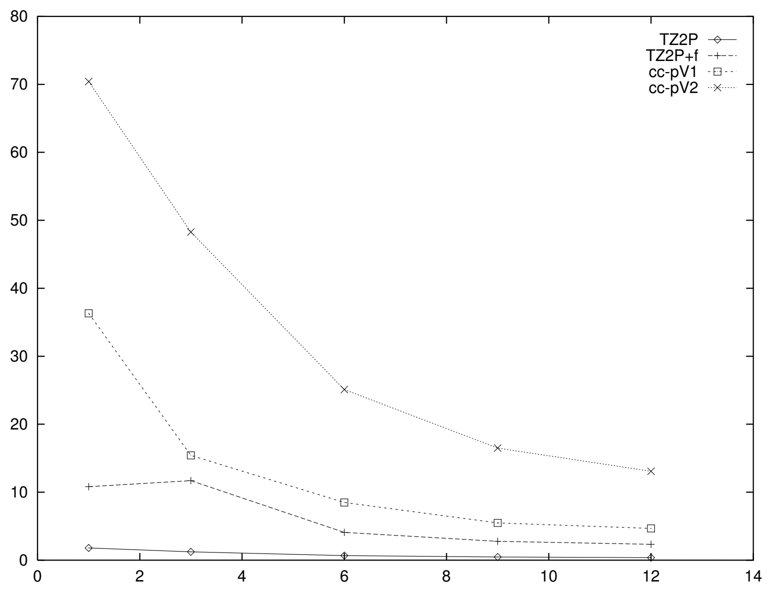

Figure 7: Run time (seconds) vs. number of processors for *rdtwo*.

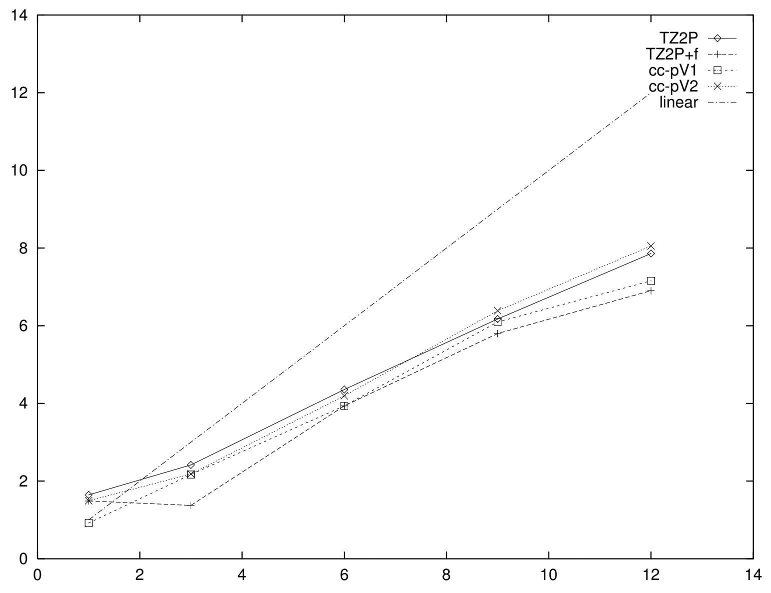

Figure 8: Speedup vs. number of processors for *rdtwo*.

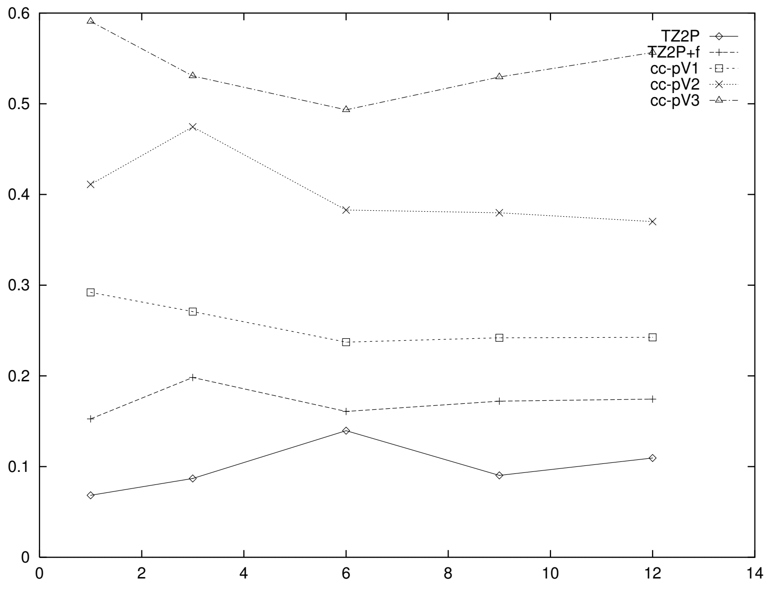

Figure 9: Run time (seconds) vs. number of processors for *mbpt*.

local memory accesses. Parallelization in this case was very useful, as this allowed more memory references to become local. Superlinear speedups (for example, a speedup of 230 on 30 processors) were observed in some of the larger test cases (in particular, cc-pV2 and cc-pV3), and run times in those cases were significantly reduced in all processor set sizes. It can be concluded, then, that memory structure plays a significant role in the applicability of parallelization to a given algorithm.

4.5 Overall Performance

We summarize our results with the total run times and speedups shown in Figures 10 and 11. Parallelization appears to be very successful in most cases, although on the larger processor sets speedup appears to level off. As mentioned above, cc-pV3 does not even include the parallelized implementation of *rdtwo*. Had we parallelized the I/O operation in *rdtwo*, this would no doubt be improved further.

For comparison purposes, we attempted to run a version that included locks around the critical section of *trans_two*, using a different lock for each element of the matrix. This machine was incapable of allocating enough locks for even the smallest test case, thus demonstrating the severe difficulties with this approach. On the KSR-2 (which uses a slightly different implementation of Cthreads) the locking algorithm was able to execute because many more locks are possible on this system; however, the total program ran an average of 2941.71 seconds for the TZ2P+f case. This was nine times slower than the sequential version and 19 times slower than the parallel version, which is clearly unacceptable.

We also timed the sequential version of the code with the sorting of TFM between *trans* and *mbpt*. The original authors of the code claimed that this reduced complexity and, by avoiding the many additional memory references, saved time. This was not what we observed; TZ2P ran for an average of 11.13 s; TZ2P+f ran 66.95 s on average, and cc-pV1 ran for 198.7 s, significantly slower than the sequential program excluding the sorting operation.

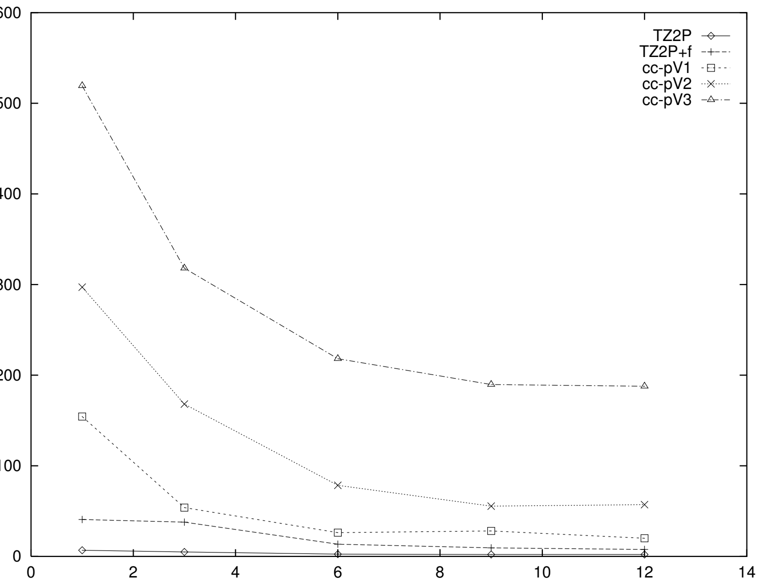

Figure 10: Run time (seconds) vs. number of processors for the whole program.

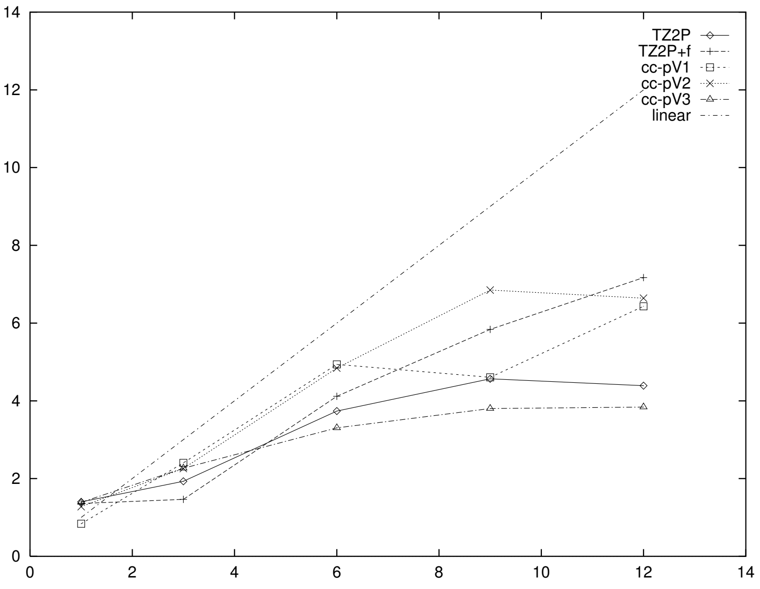

Figure 11: Speedup vs. number of processors for the whole program.

5 Future Directions

So far our efforts have been focused on applying the optimistic computation technique to a single application. We believe that this algorithm represents a unique approach to addressing the limitations of shared memory contention in specific instances. This particular application is comprised of many different components, each with its own limitations, and our approach was only applied to a single segment of the code. We would like to begin work on a new application whose behavior would better illustrate the usefulness of our algorithm.

It would also be of great interest to study the extension of this technique. Currently our approach is only useful when there are exactly two updates to any particular memory location. Straightforward approaches to allow three or more writes without a lock would involve significant memory demands, undermining the potential benefits of the algorithm. More creative approaches are necessary to apply the principles of optimistic computation to similar applications employing large data sets.

6 Conclusions

This paper presents an optimistic technique for avoiding synchronization on shared data in parallel multiprocessor applications. This technique permits computations to be executed without explicit synchronization, but then detects and corrects for shared memory access errors when they occur. A more conservative, locking approach implemented for comparison purposes either fails to run or exhibits low performance on modern shared memory platforms, due to its excessive use of operating system resources. This suggests that optimistic approaches like the one presented in this paper may be of importance for other scientific applications, as well.

The novel optimistic execution technique is used in a parallel implementation of the calculation of the second-order Møller-Plesset perturbation theory energy for closed-shell molecules. We examine the performance of the main component, *trans*, for increasingly large basis sets on increasing numbers of processors. Performance analysis indicates that for smaller systems, the parallelization is relatively expensive, particularly as larger and larger processor sets result in more memory conflicts, causing detection and correction to occur frequently. Peak behavior is observed for most test cases at a processor set size of 9, although improvements are possible beyond that number depending on the number of repetitions required in the *trans* component of this code. Although scalability is limited, performance of the optimistic approach is quite good for a reasonable number of processors.

Acknowledgements

We would like to thank T. Daniel Crawford and the members of the Center for Computational Quantum Chemistry at the University of Georgia for their help in initiating this project and their continued support in dealing with the physical chemistry aspects of the programs.

References

[CCD+84] E. Clementi, G. Corongiu, J. H. Detrich, H. Khanmohammadbaigi, S. Chin, L. Domingo, A. Laaksonen, and H. L. Nguyen. Parallelism in computational chemistry: Applications in quantum and statistical mechanics. In E. Clementi, G. Corongiu, M. H. Sarma, and R. H.

Sarma, editors, *Structure and Motion: Membranes, Nucleic Acids, and Proteins*, pages 49–85. Adenine Press, New York, 1984.

[Cle85] Enrico Clementi. Ab initio computational chemistry. *Journal of Physical Chemistry*, 1985:4426–4436, 1985.

[Cot90] Albert F. Cotton. *Chemical Applications of Group Theory.* John Wiley and Sons, New York, third edition, 1990.

[Dun71] Thom H. Dunning. Gaussian basis functions for use in molecular calculations. III. Contraction of $(10s6p)$ atomic basis sets for the first-row atoms. *Journal of Chemical Physics*, 55:716, 1971.

[Dun89] Thom H. Dunning. Gaussian basis sets for use in correlated molecular calculations. I. the atoms boron through neon and hydrogen. *Journal of Chemical Physics*, 90:1007, 1989.

[GBD$^+$94] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM 3 User's Guide and Reference Manual.* Oak Ridge National Laboratory, Oak Ridge, Tennesssee, 1994.

[HB84] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing.* McGraw-Hill Book Company, New York, 1984.

[Huz65] Sigeru Huzinaga. Gaussian-type functions for polyatomic systems. I. *Journal of Chemical Physics*, 42:1293, 1965.

[JSS$^+$94] C. L. Janssen, E. T. Seidl, G. E. Scuseria, T. P. Hamilton, Y. Yamaguchi, R. Remington, Y. Xie, G. Vacek, C. D. Sherrill, T. D. Crawford, J. T. Fermann, W. D. Allen, B. R. Brooks, G. B. Fitzgerald, D. J. Fox, J. F. Gaw, N. C. Handy, W. D. Laidig, T. J. Lee, R. M. Pitzer, J. E. Rice, P. Saxe, A. C. Scheiner, and H. F. Schaefer. *PSI 2.0.8.* PSITECH, Inc., Watkinsville, GA 30677, U.S.A, 1994.

[MD95] Antonio M. Márquez and Michel Dupuis. Parallel computation of the MP2 energy on distributed memory computers. *Journal of Computational Chemistry*, 16:395–404, 1995.

[MEG94] Bodhisattwa Mukherjee, Greg Eisenhauer, and Kaushik Ghosh. A machine independent interface for lightweight threads. *Operating Systems Review of the ACM Special Interest Group in Operating Systems*, page 33, 1994.

[MP34] C. Møller and M. S. Plesset. Note on an approximation treatement for many-electron systems. *Physical Review*, 46:618, 1934.

[Roo51] C. C. J. Roothaan. New developments in molecular orbital theory. *Reviews in Modern Physics*, 23(2):69, 1951.

[SO89] Attila Szabo and Neil S. Ostlund. *Modern Quantum Chemistry: Introduction to Advanced Electronic Structure Theory.* McGraw-Hill, New York, 1989.

[SvL83] V. R. Saunders and J. H. van Lenthe. The direct CI method. a detailed analysis. *Molecular Physics*, 48:923, 1983.

[WD88] John D. Watts and Michel Dupuis. Parallel computation of the Møller-Plesset second-order contribution to the electronic correlation energy. *Journal of Computational Chemistry*, 9:158–170, 1988.

[Wil87] S. Wilson. Four-index transformations. In *Methods in Computational Chemistry*, volume 1. Plenum Press, 1987.