EPA/600/A-97/017

Optimizing a Photochemical Air Quality Model Gas-Phase Chemistry Solver for Parallel Processing

Cheung (David) Wong#, Jeffrey O. Young*, and Edward W. Davis#

Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206

* Atmospheric Sciences Modeling Division
Air Resources Laboratory
National Oceanic and Atmospheric Administration
Research Triangle Park, NC 27711
On assignment to the National Exposure Research Laboratory,
U. S. Environmental Protection Agency

ABSTRACT

In a testbed model based on EPA's Regional Oxidant Model, the Quasi-steady State Approximation (QSSA) gas-phase chemistry solver dominates the computation, which is typical of Eulerian grid cell air quality models. We report results from optimizing the testbed solver on a Cray T3D parallel system. Since the QSSA calculations are spatially independent, an effective data decomposition is to parallelize on the grid cells. We use a simple processor mapping to assign a block of grid cells to each T3D processing element (PE). Local performance in each PE is a key issue. To minimize execution time, programming optimization techniques, such as cache collision avoidance and loop unrolling, have been applied. Proper application of such techniques has a significant effect on performance, leading to improved speedup as compared with simply relying on optimizing compilers. The testbed's simulated chemical reactions proceed at widely varying rates, resulting in a stiff system, which presents a load balance problem. That is, PEs become idle while remaining PEs complete their tasks within a simulation time step. Based on experience with optimizing the QSSA on Cray vector supercomputers, a dynamic task allocation approach to deal with load imbalance is described and performance results are given.

INTRODUCTION

Demands for more accurate representations of the physical processes and higher resolution have increased the computational complexity of emerging air quality models (AQM's). Current models have been implemented on vector architectures with good results, but future models will require much higher performance, necessitating the use of massively parallel processors (MPP's).

Among the various physical processes simulated by air quality models, most of the computational effort is spent in solving the chemical transformations of atmospheric gaseous species. Thus efforts to significantly improve the computational performance of AQM's implies focusing on the chemistry solvers.

TESTBED CORE MODEL

Successful implementation of AQM's on MPP's requires major shifts in programming paradigms. In an effort to understand the interplay between computational algorithms and MPP architectures, we have developed a testbed core model (TCM) to study the effects of algorithms and data distributions on performance. In particular, we have focused primarily on the Cray T3D MPP [1], [2]. This high performance parallel processing system uses up to 2048 DEC Alpha 64-bit RISC microprocessors.

The TCM, based on the Regional Oxidant Model (ROM) [3], [4], simulates horizontal transport, vertical exchange between three modeled layers with emissions and deposition, and gas-phase interactions. We use a test dataset, provided by the ROM system input processors [4], for a 64 column (longitude) by 52 row (latitude) domain in the northeast U.S. for one high-ozone 24-hour period. Grid cell resolution is 1/4° longitude by 1/6° latitude. The TCM was initially optimized for a vector architecture Cray C90 to provide reference code, output data, and performance statistics in order that results from the T3D implementation could be verified and compared.

Gas Phase Chemistry Solver

Characterization of gas-phase chemical reactions in air pollution models results in stiff systems of ordinary differential equations (ODE's). The ODE system is stiff because some reactions are very fast, on the order of fractions of seconds, while others have a time scale on the order of days. Stiffness in the ODE system causes numerical problems because prohibitively small integration time steps are needed to capture the small time scales and keep the method stable. In addition, the reaction rates may vary widely due to photolytic effects and system forcing from chemical emissions.

The TCM uses an explicit, quasi-steady state approximation (QSSA) solver, which is summarized in [5]. It solves the RADM2 chemical mechanism [6], which represents a current state-of-the-science mechanism that has been used extensively in modeling. On a cell-by-cell basis, the solver determines a time step with which to integrate the ODE system. In the TCM the time step can range from one second for the stiffest cells up to five minutes.

The solver iterates on the cell's time step until it accumulates five minutes, following which vertical exchanges between the model layers takes place. This sequence of solver iterations followed by vertical exchange repeats until the half-hour horizontal advection time is reached. Then, following a fresh infusion of transported trace gas chemicals, the whole algorithm is repeated over the scenario's time period. Figures that appear later in this paper show evidence of the five-minute and half-hour timings.

Implementation

For the TCM implementation, the 3-dimensional array containing the concentration field is decomposed in spatial dimensions, i.e. row and column. Decomposition and distribution of data can be accomplished using block, cyclic block, or cyclic schemes along row and column dimensions. The distribution scheme is selected to minimize the combination of computation and communication time, that is, to minimize total execution time. The selection depends on the number of processors to be used, their logical configuration, and interprocessor communication patterns, as described in a study of data distribution on parallel machines [7]. When the data are passed into the gas-phase chemistry solver, the 3-D sub-domain in each processor is collected into a one-dimensional array. Because of this data structure dimension reduction, the method of decomposition in other parts of the model is transparent to the user and to the gas-phase chemistry code.

The main calling routine invokes first the horizontal transport, followed by the chemistry/vertical-exchange routines. By using a technique called "shared-to-private coercion", it ensures data locality of the concentration field on each processor. That is, although routines are called in parallel on the set of processors, each processor references, through its argument list, only its portion of the data for processing.

With this decomposition, the chemistry and other processes are assigned one or more sub-domains in which to work, and each sub-domain belongs to a separate processor. Thus an important focus is the optimization of the chemistry on individual DEC Alpha microprocessors within the T3D system.

INDIVIDUAL PROCESSOR OPTIMIZATION

When designing algorithms for a parallel processor system, or when porting scientific applications to such a system, it is desirable to identify independent parts of a problem for assignment to independent processors. The performance of these independent chunks of work on a single processor is then a key issue in total system performance. This section describes two optimization techniques that apply to individual processors and gives experimental results on the impact of the techniques. We further report significant performance improvement for these techniques combined with additional optimization strategies for the QSSA chemistry solver.

Cache Collision Avoidance

As the fastest level in the memory hierarchy of a shared memory multiprocessor, the cache memory connected to an individual processor has an important impact on performance. Caches hold copies of small groups of words from DRAM, or main, memory in a system. In the T3D, the cache associated with each processor has 256 direct-mapped lines of four words. Line i of main memory maps only to cache memory line i mod 256. When an application attempts to access memory locations which are apart by multiples of 256 lines in the address space, collisions on use of the cache occur and cause time consuming cache misses and updates. Cache misses can result in a serious time penalty for a memory access.

Here we describe a scheme that uses data offsets to produce a favorable mapping between main memory and the cache when there are arrays of consecutively accessed data. This situation occurs frequently, as in the chemistry solver that motivated this research. The scheme improves single processor performance in a parallel processing system. In addition, the scheme allows applications to be programmed to run on various numbers of processors without recompiling.

A formal description of the scheme for one-dimensional data arrays is given, followed by a procedure for determining the offset and an example. Some notation is required. Let each processor in the parallel system contain a direct-mapped cache of total size c words, with w words per line. The number of lines is c/w. Let the number of processors to be used for a given execution be any number from the set $\{n_1, n_2, ..., n_p\}$. Let v represent the number of one-dimensional data arrays declared in the application and let S represent the number of elements of each of the v arrays. If the application arrays are distributed over the set of processors as local arrays, then $s(n_i)$, $n_i \in \{n_1, n_2, ..., n_p\}$, represents the size of the local arrays. Note that his size is a function of the number of processors.

Theorem: There exists an offset k such that cache collisions will not occur in any sequential access iterative loop in a program if the following criterion is met for all n_i:

$$1 + \frac{c}{\gcd(s(n_i) + k, c)} > v, \quad \text{where gcd is the greatest common divisor} \qquad (1)$$

A proof of this theorem is in [7] where, also, the scheme is extended to two dimensional arrays and to handling a mixture of one and two dimensional arrays.

The theorem only indicates the existence, or lack thereof, of an appropriate offset. We now show a procedure to calculate the offset, based on this theorem but specifically for the T3D architecture. In the set $\{n_1, n_2, ..., n_p\}$, each n_i is a power of two and $n_{i+1} = 2n_i$. We further assume that c is a power of two, as is typical of commercial systems. The procedure is given by the iterative construct below. It selects the smallest potential value for k and tests it against the criterion from the theorem. If the criterion is not satisfied, it selects successively larger values of k. If the set of choices for k is exhausted without finding an appropriate offset, there is none.

found = false
For $\beta = 1$ *to* c/w *do while* $\neg$*found*
    $k = \beta * w$
    $\alpha = max\ \{gcd(s(n_i) + k, c),\ 1 \le i \le p\}$
    If $\lfloor c/(v-1) \rfloor > \alpha$ *then*
        found = true /* *k is an appropriate offset* */
End for.

As an example of this procedure, consider a problem with four one-dimensional arrays of size S = 128. Let the processor set be {4, 8, 16} with each processor having a direct-mapped cache of size 32 with four words per line. That is, $c = 32$ and $w = 4$. Therefore $s(4) = 32$, $s(8) = 16$, and $s(16) = 8$.

The first statement in the loop, with $\beta = 1$, assigns $k = 4$.

In the second statement, $\alpha = max\{gcd(32 + 4, 32), gcd(16 + 4, 32), gcd(8 + 4, 32)\} = 4$.

Since $\lfloor 32/(4-1) \rfloor > 4$, the criterion of the conditional statement is satisfied and $k = 4$ is an appropriate offset. Continuing this example, Figure 1 illustrates both the original mapping of arrays to cache and the conflict free mapping. We assume that the four arrays, labeled A, B, C, and D, are declared in contiguous locations in memory. In the original mapping, without offsets, corresponding elements of different arrays map to the same cache lines, causing collisions for operations involving these elements. With an offset of four, the mapping is collision free. Figure 2 is similar for cases of using eight or 16 processors. It shows the mappings in an abstract way by indicating where the initial element of each array would be stored. The scheme determined that an offset of four would be collision free for all cases, as is shown in the figures.

This scheme was used for the gas-phase chemistry application running on processor configurations from the set {8, 16, 32, 64, 128, 256}. The QSSA code has 11 one-dimensional local data arrays and 11 with two dimensions. On the T3D there is no direct mechanism to record cache misses, thus we used execution time as the performance measurement. Table 1 shows the performance of the solver with its original code and then with data structure offsets as determined by this scheme to avoid collisions. Each experiment was run for ten iterations. Improvement is seen to range from about 14% to 21%. The relative improvement remains good as the number of processors increases.

The notion of using offsets to avoid cache collisions has been suggested before. This research has analyzed the situation and developed a robust method to determine an appropriate offset when there are

large numbers of arrays with various dimensions and when the code is intended for running on various numbers of processors without recoding and recompiling. The scheme renders the smallest possible offset.

Cache collision avoidance can be used in conjunction with other performance enhancing techniques such as loop unrolling, described next.

Loop Unrolling

The importance of loops and their potential for parallelism has led to many research efforts having the goal of improving performance. One technique, loop unrolling, is a standard optimization technique. It has been incorporated into compilers such as CFT77 on Cray vector systems and has also been adopted for parallel machines. One could assume there is nothing to gain through explicit programmer coded unrolling, however, our experience has shown that manual unrolling enhances performance of parallel processor systems more effectively than relying solely on compiler optimization.

The essential idea is that the body of an unrolled loop contains n copies of the original loop body, and therefore represents n iterations of the loop. Unrolling has to be done in such a way as to maintain the same computational work as the original but with fewer iterations. The logical extension of the idea is to completely unroll loops and eliminate all related overhead. However, performance benefits are limited by instruction cache size and available registers. If the unrolled loop instruction body exceeds the cache size, identical code segments will have to be reloaded as execution proceeds through the straight line sequence. Selecting the optimal depth of unrolling requires insight to the behavior of the particular code on the processor.

We conducted experiments to assess the benefit of using this technique for the T3D. Five kernels and an application code were tested with various depths of unrolling. Currently the compiler does not perform unrolling automatically, but provides a directive to allow the programmer to specify the depth. Multidimensional arrays are processed with nested loops. Unrolling was done along different dimensions of the arrays, producing somewhat different performance results but the same general conclusions. Empirical results are presented in Table 2 for the six codes with various depths of unrolling.

It can be seen that loop unrolling has a significant, but non-uniform, impact on performance. There is no specific guide to the choice of depth of unrolling. Matrix multiplication, for example, shows that increasing the depth from five to seven led to a decrease in performance. This simply indicates the need for programmer experimentation with the code to determine suitable depth and dimension information. Then, when incorporated in production codes, unrolling can provide speedups in the range of two to four. Further empirical results and discussion are in [7]. We applied manual loop unrolling to the TCM with performance benefits comparable to those for the experimental kernels.

Performance Results with Programmer Optimization

Additional optimization techniques were employed toward the goal of minimizing total execution time. For comparison purposes, a baseline of parallel performance for QSSA was obtained by porting a highly vectorized version to the T3D. Block decomposition, in which each processor is assigned a contiguous, equal fraction of the total data, was used to assign multiple geographical cells to each processor. Using compiler optimization as the only means of performance enhancement, each experiment was run for ten iterations.

Table 3, in the left-hand columns, shows execution time results for compiler-optimized code. It also indicates speedup relative to execution time of 524 seconds on a single T3D processor. Memory requirements for running QSSA exceed the 64 Mbytes available with a single processor. Thus the single processor run actually required the memory of four processors, incurring a data access time penalty and lengthening execution time. Note that the speedup has very modest improvement as the number of processors increases. It does not offer any promise for cost effective scaling to large numbers of processors.

Programmer optimization, based on knowledge of the application and the architecture, has a dramatically different result. The techniques mentioned earlier, cache collision avoidance and loop unrolling, were used to improve performance. In addition, performance was enhanced by distributing data in an interleaved-block partition scheme to improve load balance, localizing data access to minimize communication and synchronization, interchanging dimensions of the two-dimensional arrays so that data reuse is possible, coercing shared data to private, effectively using the instruction cache, and prefetching data.

The net effect of these techniques is also shown in Table 3, in the right-hand columns, for the same problem parameters as was used for pure compiler optimization runs. With programmer optimization, execution time is greatly reduced and the performance scales very well from eight to 128 PEs. Recall that the single processor case incurred a data access time penalty due to the use of memory in four PEs. The

long execution time for a single processor leads to speedup figures that exceed the number of PEs. The important thing is that execution time was cut in half with each doubling of the number of PEs. Table 3 also shows the speedup due simply to programmer optimization by comparing execution times from the compiler optimization and programmer optimization columns. This shows that knowledgeable programming can yield dramatic speedup on a fixed number of PEs.

Next we address the larger system issue of processor workload balance.

SYSTEM LOAD BALANCE

Effective task management is one of the major issues affecting performance of parallel computing systems. Failure to manage the assignment of tasks to PEs can lead to large differences in the computational load at each PE, which results in poor performance. The testbed's simulated chemical reactions proceed at widely varying rates, resulting in a stiff system, which presents a load balance problem. That is, PEs become idle while remaining PEs complete their tasks within a simulation time step. There are two approaches to task allocation that attempt to provide balance in use of resources. A static approach determines a task allocation before run time and the allocation pattern remains the same for the rest of the program execution. However, in a dynamic approach, allocation is determined during run-time and the allocation may change as program execution progresses.

Static allocation is the simplest strategy to balance the work load based on *a priori* knowledge of computational intensity of each cell in a data domain. Under the ownership rule, in which data is owned by a processor with rights to update it, adopting a static scheme is equivalent to determining an appropriate data decomposition which minimizes computation and communication costs.

Domain decomposition in a static scheme renders rectangular sub-domains. Generally speaking, the dimensions of the data domain are not evenly divisible by the number of processor assigned to the dimensions. By modifying the decomposition pattern to non-rectangular shapes, better load balance can be achieved [8]. Current high-level parallel programming languages only support partitioning data into rectangles. Non-rectangular decomposition can be obtained through software implementations[8], [9].

A static approach is not well-suited for applications, such as an air quality model, that possess non-uniform execution behavior among data points. The remainder of this section reports on static and dynamic approaches to load balancing for the testbed model.

Static Allocation Applied to the TCM

The distribution of stiff cells over a domain is irregular and changes over the course of the simulation because of diurnal photolytic effects and emissions effects. Figure 3 displays computational behavior of the testbed domain at two different scenario times. The charts indicate the frequency with which the solver had to iterate for the counts along the x-axis. The higher the count, the stiffer the cell. A majority of the grid cells are non-stiff. However, when a grid cell is stiff, it takes longer to complete the entire iterative process. Furthermore, the number of stiff cells, their distribution, and the stiffness change as simulation progresses. Hence it is impossible for a static approach to determine a perfectly balanced work distribution.

Figure 4 shows an imbalance in the work when a static decomposition scheme was used with eight processors. Individual processors took differing amounts of time to complete their work, as shown by the number of clock cycles. Individual bars within the groups of bars in Figure 4 show the work load in each processor. The rightmost bar among each group of bars is the average execution time and indicates the level of work in each processor under a perfect task distribution. By charting the maximum and minimum work load within a set of allocated processors, as in Figure 5, a lack of balance is emphasized.

Dynamic Approaches

Numerical solution of the sequential model is based on an operator splitting technique. The model simulation is divided into three components: horizontal transport calculation, vertical flux calculation, and gas-phase chemistry computation. Calculation of the first two components involves communication in the spatial dimensions. However, the gas-phase chemistry computation is spatially independent. Hence, in the implementation, a predetermined two-dimensional spatial decomposition was used which balances the computation and communication cost. When the executing code enters the chemistry solver, the three-dimensional domain is collected into a one-dimensional array. Thus, the actual domain decomposition is transparent to the solver.

Within the chemistry solver, each processor inherits its data from prior computational components. Since domain decomposition does not consider stiffness in each geographical cell, and the distribution of stiff cells changes with respect to time, the issue is to determine an appropriate dynamic scheme to balance the work load.

Since the TCM operates in a Single Program Multiple Data (SPMD) mode, adopting a centralized data distribution scheme is inefficient and inappropriate. An alternative is to let each processor handle its own data. If a processor finishes its work before completion of the process, it can look for extra work from other processors. An algorithm was developed that provides a mechanism to choose the next piece of work, and in addition maintains consistency of work distribution, balances the work load, and renders minimum overhead.

Recall that the three-dimensional domain is collected into a single dimension and consider the processors to be in a cyclic arrangement, each with an initial assignment of cells to processors. When a processor finishes its own work, three different ways to choose the next piece of work were considered: (1) choose from the next processor in the right-hand direction with remaining work, (2) choose from the processor which has the most remaining work, and (3) choose from a processor so that an even processor work distribution is maintained. For instance, suppose six processors (PEs) are in use and have reached a point where PEs 1 and 2 are working as a group with 16 units of work remaining, PE 3 just finished its last piece of work, PEs 4 and 5 are working as a group with 8 units of work left, and PE 6 has 12 units of work left. The first scheme will assign PE 3 to help PEs 4 and 5 as the nearest PEs to the right with remaining work. The second scheme will assign PE 3 to help PEs 1 and 2 since they have most remaining work. However, the third scheme will assign PE 3 to help PE 6 since it has the highest remaining workload. With the last scheme, each of the three work groups has two PEs assigned which evenly distributes available resources over the remaining work.

Based on a small scale study, there is virtually no difference in performance between the three approaches in terms of balancing the work load. The first scheme was adopted because it was the simplest to implement. Effectiveness of the method is shown in Figure 6 where maximum and minimum workload bars are nearly the same in each five-minute time interval, as compared with Figure 5 where maximum workloads exceed the minimum by a noticeable amount in every time interval.

Dynamic load balancing implies the use of a method to manage scheduling, such as a critical section, which can be accessed by only one PE at a time, resulting in an attendant overhead. There are several ways to cut down critical section overhead. One approach is to have a processor work on n grid cells at a time. By intuition, the larger n is, the less overhead will be incurred. However, a large n also reduces the ability to balance the load. In an experiment with n having values of 1, 5, and 10, there was no significant difference in execution time, as shown in Figure 7. This indicates that critical section overhead is very small in comparison to computational work.

A second approach is to defer activation of scheduling and critical section use until one processor finishes all its assigned work. Experimental data again shows no significant advantage in performance using this approach.

A third approach is to eliminate the critical section completely. Based on the chemical species concentration, chemical reaction rate, and production and loss rate, an approximation of computational intensity in each grid cell can be computed easily. In other words, the distribution of stiffness can be determined so that work can be allocated evenly. However, the trade off is extensive data movement. For example, suppose stiff cells cluster heavily in processor 1. In order to maintain an even distribution of work, that processor must move a portion of its data to processor 2, which will in turn move data to next neighbor processor. Finally, a new approach under development minimizes data movement by breaking the ripple effect just described. In this approach, data is moved from processor 1 to processor 3 directly. A comparison between the approach of using a critical section with each grid cell ($n=1$) and the approach which eliminates critical sections entirely has shown that although the difference is very small, this final approach, with computed workloads, is better.

CONCLUSION

This paper has reported several approaches for improving performance when running air quality models on parallel processors. Techniques were used to optimize performance of individual processors and to consider system issues of load balance. They provided considerable speedup over optimizing compilers and other system software and program development tools. At the present time, compilers and tools for optimizing code are not competitive with the effort of knowledgeable programmers. Techniques, such as the cache collision avoidance reported here, can be incorporated into tool sets to assist programmers.

Our experience indicates that air quality modeling is amenable to parallel processing. There is a natural parallelism in the grid cell formulation of the models. For many applications, the need to communicate between processors imposes a serious communication overhead on total processing time. For this application, the computationally intensive gas-phase chemistry is spatially independent and parallelizes in an efficient way.

ACKNOWLEDGMENTS

We are pleased to acknowledge the provision of Cray T3D computer system time and other resources by the North Carolina Supercomputing Center, Research Triangle Park, NC and by Cray Research, Inc. We are also grateful for T3D programming and technical assistance from Vance L. Shaffer of Cray Research.

The research reported in this paper has been funded in part by the U.S. Environmental Protection Agency under cooperative agreement CR822076-01-2 to North Carolina State University.

DISCLAIMER

This paper has been reviewed in accordance with the U.S. Environmental Protection Agency's peer and administrative review policies and approved for presentation and publication. Mention of trade names or commercial products does not constitute endorsement or recommendation for use.

REFERENCES

1. Cray T3D Technical Summary, Cray Research, Inc., 1993.

2. Cray T3D Applications Programming, TR-T3DAPPL E Cray Research, Inc., 1994.

3. Lamb, R.G. *A Regional Scale (1000km) Model of Photochemical Air Pollution. Part I -- Theoretical Formulation*, EPA-600/3-83/035; U.S. Environmental Protection Agency: Research Triangle Park, NC, 1983.

4. Lamb, R.G. *A Regional Scale (1000km) Model of Photochemical Air Pollution. Part II -- Input Processor Network Design*, EPA-600/3-84/035; U.S. Environmental Protection Agency: Research Triangle Park, NC, 1984.

5. Young, J.O., Sills, E.D., and Jorge, D.A. *Optimization of the Regional Oxidant Model for the Cray Y--MP*, EPA/600/R-94/065; U.S. Environmental Protection Agency: Research Triangle Park, NC, January, 1993.

6. Stockwell, W.R., Middleton, P., and Chang, J.S. "The Second Generation Regional Acid Deposition Model Chemical Mechanism for Regional Air Quality Modeling", J. Geophys. Res., **1990** 95, 16343-16367.

7. Wong, C.D. *Performance Enhancement Techniques and Methodologies for Highly Parallel Processing Systems in Scientific Computational Applications*; Ph.D. Dissertation, Dept. of Computer Science, North Carolina State University: Raleigh, NC, 1996.

8. Liu, J., Saletore, V.A., and Lam, Y.B. "Partitioning of Arrays for High Performance", *Proceedings of the ISCA International Conference on Parallel and Distributed Computing and Systems*: Louisville, KY, October 14--16, 1993; pp 128-131.

9. Snyder, L. and Socha, D. "An Algorithm Producing Balanced Partitioning of Data Arrays", *Proceedings of the Fifth Distributed Memory Computing Conference*: 1990; pp 884-893.

TABLES

Table 1. Impact of cache collision avoidance on the QSSA solver.

	Execution time, in seconds					
Number of Processing Elements	8	16	32	64	128	256
Original code	14.58	7.61	3.62	1.90	0.91	0.47
Code with collision avoidance offsets	12.53	6.25	3.12	1.55	0.77	0.37
Improvement, %	14.1	17.9	13.8	18.4	15.4	21.3

Table 2. Impact of programmer-directed loop unrolling on six code segments.

Code segment	Unrolling: depth, dimension	Array sizes	Performance in MFLOPS	Speedup due to unrolling
Shallow water model, loop 100	none	256 by 256	8.12	-
	4		12.33	1.5
Level 2 BLAS: DSPR2	none	352 by 352	22.32	-
	4, in i dim		31.92	1.4
	4, in j dim		32.74	1.5
	11, in i dim		49.09	2.2
	11, in j dim		35.40	1.6
Level 2 BLAS: DSPMV	none	352 by 352	9.24	-
	4, in i dim		19.83	2.1
	11, in i dim		31.54	3.4
Livermore Fortran kernel 18	none	32 by 542	14.83	-
	6, in j dim		15.87	1.1
Matrix multiplication	none	210 by 210	6.71	-
	5, in k dim		28.66	4.3
	7, in k dim		25.69	3.8
Matrix column variance	none	352 by 352	25.94	-
	4, in i dim		38.72	1.5
	11, in i dim		41.49	1.6

Table 3. Performance of QSSA on Cray T3D. A comparison of compiler versus programmer optimization.

Number of processors	Compiler-optimized QSSA code		Programmer-optimized QSSA code		Speedup due to programmer optimization
	Execution time in seconds	Speedup over one PE	Execution time in seconds	Speedup over one PE	
8	107.8	4.9	16.17	32.4	6.6
16	82.7	6.3	8.14	64.4	8.2
32	70.7	7.4	4.04	129.7	17.5
64	65.3	8.0	2.01	260.7	32.6
128	63.8	8.2	0.99	529.3	64.5

FIGURES

line 1	A(1)	A(2)	A(3)	A(4)
line 1	B(1)	B(2)	B(3)	B(4)
line 1	C(1)	C(2)	C(3)	C(4)
line 1	D(1)	D(2)	D(3)	D(4)
line 2	A(5)			
line 3				
line 4	•			
line 5		•		
line 6			•	
line 7				•
line 8				A(32)

(original)

line 1	A(1)	A(2)	A(3)	A(4)
line 2	B(1)	B(2)	B(3)	B(4)
line 3	C(1)	C(2)	C(3)	C(4)
line 4	D(1)	D(2)	D(3)	D(4)
line 5		•		
line 6			•	
line 7				•
line 8				A(32)

(with offset: collision free)

Figure 1. Mapping arrays to cache with four processors. Conflicts occur when corresponding elements of different arrays map to the same cache lines, as shown in line 1 of the original mapping.

line 1	A,C	A
line 2		
line 3		C
line 4		
line 5	B,D	
line 6		B
line 7		
line 8		D

(original) (with offset)

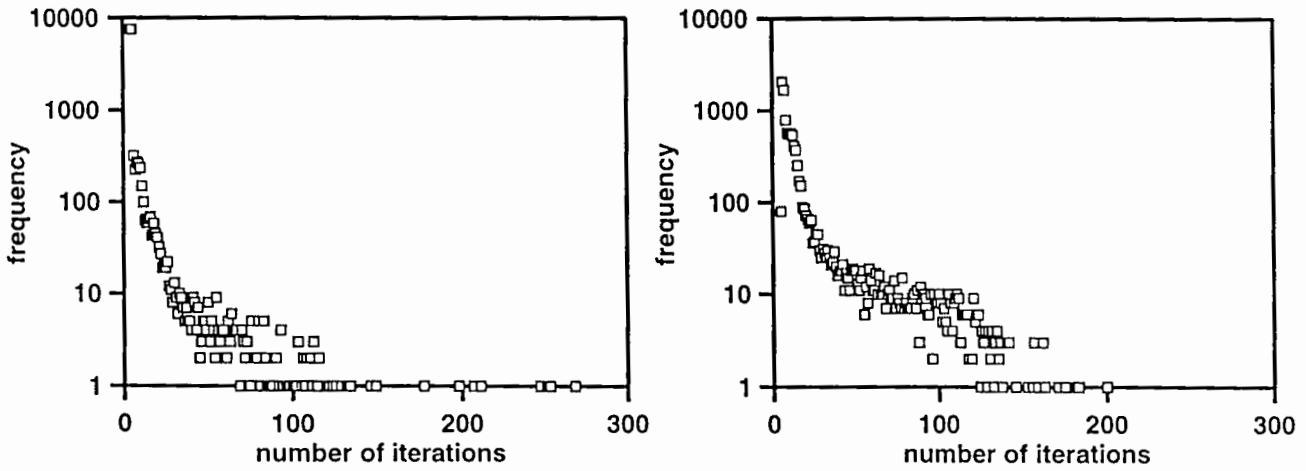
(a)

line 1	A	A
line 2		D
line 3	B	
line 4		B
line 5	C	
line 6		
line 7	D	C
line 8		

(original) (with offset)

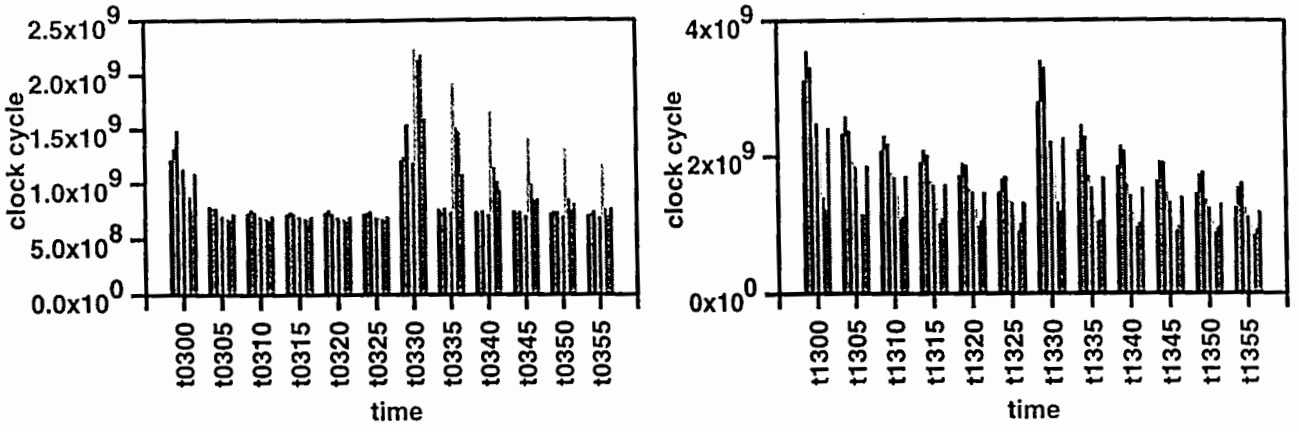
(b)

Figure 2. Mapping arrays to cache for (a) eight, and (b)16 processors.



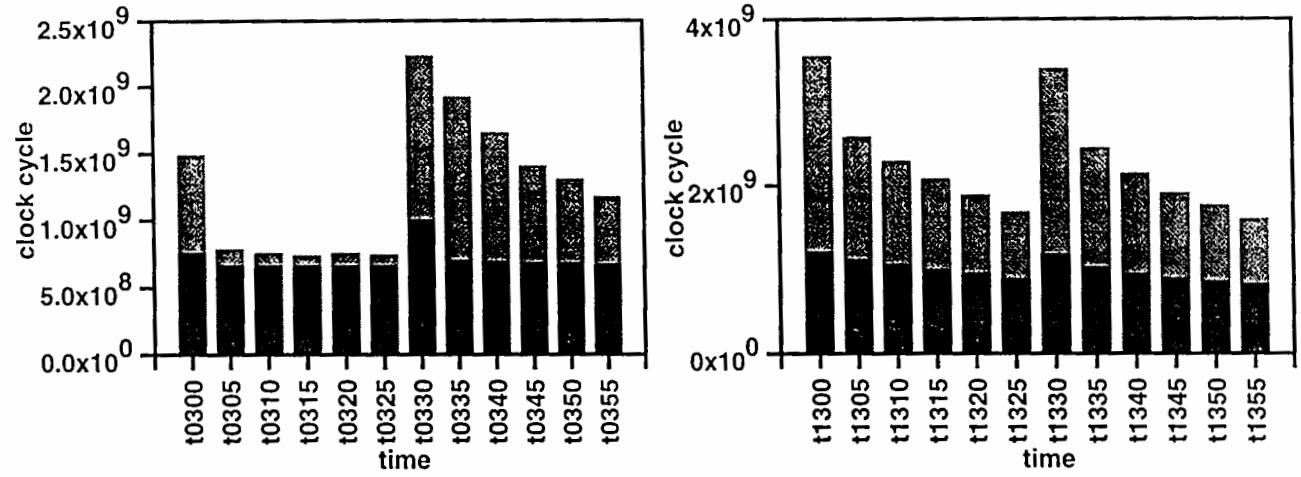
(a) (b)

Figure 3. Frequency distribution of stiff cells for a scenario at (a) 0430 and (b) 1400 hours. "Frequency" indicates a count of the cells requiring the corresponding number of iterations. Stiffer cells require more iterations.



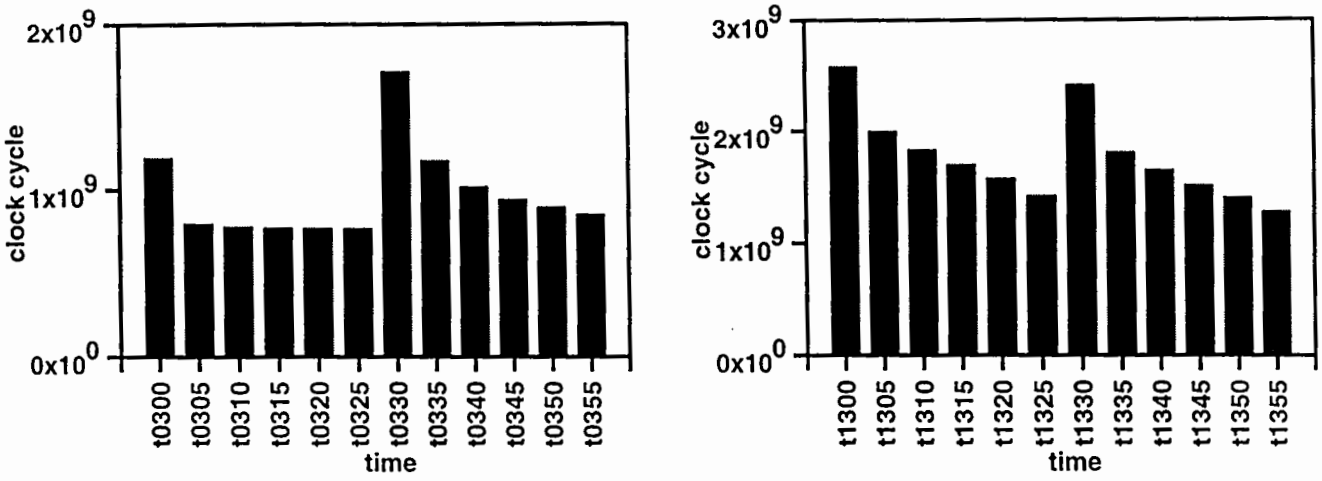
(a) (b)

Figure 4. Per processor workload, in clock cycles, for eight processors at (a) 0300 and (b) 1300 hours. The rightmost bar in each group is the average workload for the time interval.



(a) (b)

Figure 5. Maximum and minimum processor workloads, in clock cycles, starting at (a) 0300 and (b) 1300 hours. This chart is derived from Figure 4.



(a) (b)

Figure 6. Balanced processor workloads in every time interval. In contrast to Figure 5, fewer clock cycles are needed and the difference between maximum and minimum in each bar is not visible at this scale.

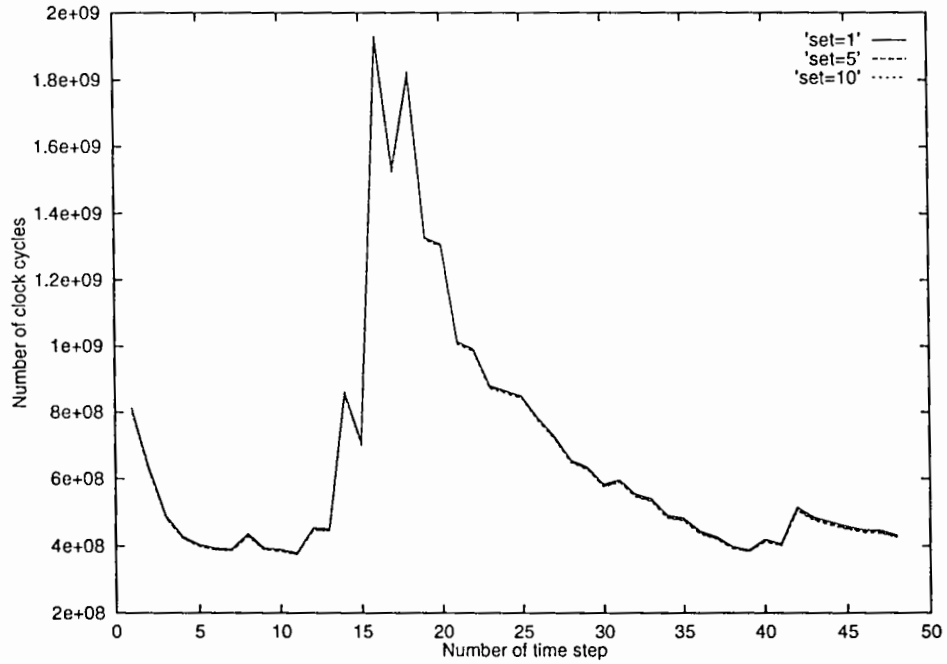


Figure 7. Critical section set size has almost no affect on overall performance. The graphs overlap nearly perfectly.

TECHNICAL REPORT DATA

1. REPORT NO.	2.	3.
EPA/600/A-97/017		

4. TITLE AND SUBTITLE	5. REPORT DATE
Optimizing a Photochemical Air Quality Model Gas-Phase Chemistry Solver for Parallel Processing	6. PERFORMING ORGANIZATION CODE

7. AUTHOR(S)	8. PERFORMING ORGANIZATION REPORT NO.
Cheung, (David) Wong [1] Young, Jeffrey O.[2] Edward W. Davis [1]	

9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. PROGRAM ELEMENT NO.
[1] Dept.of Computer Science, North Carolina State Univ. Raleigh, NC 27695-8206 [2]Atmospheric Modeling Division, National Exposure Research Laboratory, U.S. Environmental Protection Agency, Research Triangle Park, NC 27711	11. CONTRACT/GRANT NO.

12. SPONSORING AGENCY NAME AND ADDRESS	13. TYPE OF REPORT AND PERIOD COVERED
National Exposure Research Laboratory Office of Research and Development U.S. Environmental Protection Agency Research Triangle Park, NC 27711	Conference Paper, FY-97
	14. SPONSORING AGENCY CODE

15. SUPPLEMENTARY NOTES

16. ABSTRACT

In a testbed model based on EPA's Regional Oxidant Model, the Quasi-steady State Approximation (QSSA) gas-phase chemistry solver dominates the computation, which is typical of Eulerian grid cell air quality models. We report results from optimizing the testbed solver on a Cray T3D parallel system. Since the QSSA calculations are spatially independent, an effective data decomposition is to parallelize on the grid cells. We use a simple processor mapping to assign a block of grid cells to each T3D processing element (PE). Local performance in each PE is a key issue. To minimize execution time, programming optimization techniques, such as cache collision avoidance and loop unrolling, have been applied. Proper application of such techniques has a significant effect on performance, leading to improved speedup as compared with simply relying on optimizing compilers. The testbed's simulated chemical reactions proceed at widely varying rates, resulting in a stiff system, which presents a load balance problem. That is, PEs become idle while remaining Pes complete their tasks within a simulation time step. Based on experience with optimizing the QSSA on Cray vector supercomputers, a dynamic task allocation approach to deal with local imbalance is described and performance results are given.

17. KEY WORDS AND DOCUMENT ANALYSIS

a. DESCRIPTORS	b. IDENTIFIERS/ OPEN ENDED TERMS	c. COSATI

18. DISTRIBUTION STATEMENT	19. SECURITY CLASS *(This Report)*	21. NO. OF PAGES
	20. SECURITY CLASS *(This Page)*	22. PRICE