Guidelines and Computational Results for Vector Processing of Network Assignment Codes on Supercomputers

KYRIACOS C. MOUSKOS AND HANI S. MAHMASSANI

Supercomputers derive their computational performance from faster processors as well as innovations in their architecture. To take advantage of the vector processing capabilities of supercomputers, such as the CRAY X-MP series, it is necessary to modify the code to enhance its vector processing performance. These modifications can range from simple localized recoding of existing mainframe codes to devising new algorithms with the hardware's architecture in mind. In this paper, codes for the solution of two network equilibrium assignment problem formulations (Frank-Wolfe algorithm for the single-class user equilibrium problem and the diagonalization algorithm for multiple user classes with asymmetric interactions) are vectorized and tested on a CRAY X-MP/24 supercomputer. Only local vectorization by limited recoding of existing programs is performed. Guidelines are given for this purpose, and their application to the assignment codes is illustrated. The computational tests performed indicate an improvement in execution time of about 70 to 80 percent of the modified code relative to its unvectorized performance on the CRAY supercomputer. Execution of the vectorized code on the CRAY is about 22 times faster than the execution of the unmodified code on a mainframe computer. The significance of the results for research and practice is also discussed.

The network traffic assignment problem arises in connection with many transportation planning activities, including the analysis of the cost-effectiveness of capital improvement projects and the evaluation of operational planning strategies in traffic networks. Two decades of research have resulted in efficient and widely available algorithms for this problem, particularly for the case of a single class of users and no interactions across links. Such programs are routinely executed on microcomputers, though only for moderately sized networks. A review and textbook presentation can be found elsewhere (*1*). For more complicated and realistic cases, especially those involving multiple user classes and asymmetric link interactions (1–5), existing algorithms are much more demanding computationally, especially for large-scale systems. Network assignment procedures are also critical for solving the network design problem, which is an *np*-hard problem that cannot generally be solved optimally using current computational techniques.

Supercomputers offer at least an order of magnitude improvement over conventional mainframes in terms of speed and memory capabilities, and they greatly enhance our ability to solve large problems under more realistic assumptions. Supercomputers derive their high performance not only from inherently faster silicon chips, whose performance is fast approaching its quantum-mechanical limits, but also from their radically different architectures that reflect different degrees of parallelism (*6,7*). The CRAY X-MP series of supercomputers, which is used in the present study, appears to have gained the widest acceptance and accessibility in the American academic community. Its architecture provides a dimension of parallelism by using vector or matrix operations of an algorithm (vectorization). More detailed description of the hardware aspects of the CRAY X-MP that are relevant to applications programmers can be found in papers by Zenios and Mulvey (*7*) and Chen (*8*).

Compilers are generally available for the CRAY supercomputer to "vectorize" a particular code by identifying those independent portions that can be executed in parallel and sequencing the processing and task allocation accordingly. However, there are many inherently parallel activities that may have been programmed in ways intended for conventional scalar processing but that actually inhibit the vectorization capabilities of the compiler. It is therefore generally possible to take fuller advantage of the capabilities of the supercomputer's architecture by modifying, or vectorizing, the code. Three levels of vectorization can be distinguished (*7*):

1. Local software vectorization, where the program is reexamined in its parts and subroutines, and redesigned only locally, without program-wide repercussions;
2. Global software vectorization, affecting the whole implementation of the algorithm and the design of the data structures; and
3. Overall algorithm vectorization, where the solution algorithm itself is conceived to take advantage of the machine architecture.

Recently, Zenios and Mulvey (*7*) provided an example of the kinds of local modifications needed to vectorize codes for the solution of nonlinear network programs and reported related computational experience on the CRAY X-MP/24. In addition to illustrating the potential of supercomputers for solving large-scale network optimization problems, their results highlighted the need to modify the codes to achieve better vec-

Department of Civil Engineering, University of Texas at Austin, Austin 78712.

torization. The present paper presents similar information for codes to solve the traffic network equilibrium assignment problem. The principal objective is to assess the computational improvements that can be achieved by local vectorization of network traffic assignment codes, for the single-class and the two-class (with asymmetric interactions) user equilibrium problems. The computational experiments are performed on the CRAY X-MP/24 supercomputer. The results have important implications for practice in terms of the size and complexity of the problems that can be addressed and, more important, for the future development of solution approaches to the network design problem.

The next section presents general guidelines for the local vectorization of FORTRAN codes. Following a brief description of the algorithms, the application of these principles to the single-class user equilibrium assignment codes, and the corresponding computational improvements are described. Results for the two-class problem are presented next, followed by concluding comments.

CODE VECTORIZATION GUIDELINES

To develop vectorizable programs and properly exploit the supercomputer capabilities, some appreciation of the machine's architecture and characteristics is helpful (*7,8*). The CRAY X-MP consists of separate dedicated functional units for vector floating point operations, vector integer operations, and scalar integer operations, respectively. It contains eight vector and eight scalar registers where vectors and scalars, respectively, are held before and after being processed on their way from and back to the memory. Vectors are processed in a pipeline fashion; after an initial startup period the first result appears, followed by the other results, one every cycle. The Cray FORTRAN (CFT) compiler produces a code that contains vector instructions to drive the high-speed vector and floating point functional units and the eight vector registers in their specified operation. The compiler, to be on the safe side, does not attempt vectorization when it suspects certain dependencies within DO loops, even if the corresponding operations are inherently vectorizable. Another important feature of the CRAY X-MP is the abundance of memory and availability of a very high speed, large solid-state device. As such, many of the techniques typically used to reduce and carefully manage storage in programs developed for mainframe computers may actually inhibit vectorization and degrade performance on the supercomputer.

The first step in the local vectorization of a program initially developed for scalar processing is to perform a time requirements analysis to determine the time-intensive parts of the code. These should then become the primary targets of the recoding effort. A combination of code modifications and compiler directives can then be employed. This process is iterative and can be continued until the programmer is satisfied that no further meaningful improvement can be achieved. Beyond this level, additional improvements would have to be sought by higher-level vectorization, as described earlier.

The primary programming constructs that should be targeted in vectorization efforts are DO loops, where the majority of computer time expense is incurred. As already noted, the CRAY X-MP compiler automatically tries to vectorize the loops where applicable. When trying to determine whether or not to vectorize a particular DO loop, the CFT compiler checks for the existence of any dependencies within the loop. Statements that should be avoided within the DO loops, according to the UT CHPC User Services Group (*9*), include CALL statements; I/O statements; branches to statements not in the loop; statement numbers with references from outside the loop; references to character variables, arrays, or functional IF statements that may not execute because of the effects of previous IF statements; ELSE IF statements.

The guidelines presented next were followed in vectorizing the network assignment codes, based on suggestions in the publications of the UT CHPC User Services (*9*) and the San Diego Supercomputer Center (*10*):

1. Data dependencies should be eliminated; a loop will not vectorize if, for example, an array is referencing values dependent on computations in lower portions of the array in an incrementing loop. The computations cannot be pipelined.
2. Subscript ambiguities should be eliminated; try to eliminate the dependency of a subscript on a previous calculation by including the operation in the array.
3. In the case of nested loops, the one with the largest range should be assigned as the innermost loop; this would contribute the most to the overall effectiveness of the code because the inner loop is the only one that is vectorized.
4, Conditionals should be eliminated; IF THEN ELSE statements can be replaced by conditional vector merge procedures. Simple IF statements are vectorizable but might inhibit vectorization if their references lead to some of the aforementioned dependencies.
5. The loops should be unrolled to a certain depth, thereby eliminating checking for termination conditions and enforcing chaining and functional unit overlap.
6. Vectorizable loops should be separated from unvectorizable loops—in particular, separate loops that contain CALL statements or I/O statements or any of the statements mentioned previously that are independent of the other computations within the loop.

Before describing the application of these rules to the network assigment codes considered in the study, the basic steps of the algorithms for the single-class user equilibrium and the multiclass user equilibrium with asymmetric costs problems are presented.

REVIEW OF THE NETWORK EQUILIBRIUM ALGORITHMS

Given a known matrix of origin-destination flows, a network of directed links connecting nodes, and link performance functions that describe the dependence of link costs on the corresponding link flows, the single-class user equilibrium algorithm solves for the flows onto the individual links of the network so as to achieve certain equilibrium conditions whereby no driver can improve her travel time by unilaterally switching routes. Exact solution algorithms for the single-class user equilibrium problem are based on Beckman's equivalent mathematical programming formulation (*11*), which can be solved by any of several nonlinear optimization techniques. The most widely used algorithm for its solution is based on the Frank-Wolfe or convex combinations method. This algo-

rithm is well documented, and a detailed presentation can be found in a paper by Sheffi (*1*). A brief overview is presented here.

The algorithm consists of an iterative procedure that, at each iteration, first finds a search direction by solving a linearized approximation, then solves for the optimal move size along that direction. The efficiency of the algorithm derives from the fact that the direction-finding step is equivalent to performing an all-or-nothing assignment. The latter requires the repeated application of a shortest path routine, which is the principal computationally demanding element of the code. An additional source of computational cost is the line search to find the optimal move size along a particular direction and the computation of the relatively complicated nonlinear travel cost (link performance) functions. Letting $t_a(.)$ denote the link performance function for link a, the principal steps of the algorithm can be summarized as follows:

STEP 0: Initialization. Perform all-or-nothing assignment based on the free flow travel times $t_a = t_a(0)$, $\forall a$; This yeilds the set of link flows $\{X_a^1\}$. Set counter $n = 1$.

STEP 1: Update. Set $t_a^n = t_a(X_a^n)$, $\forall a$.

STEP 2: Direction finding. Perform all-or-nothing assignment based on $\{t_a^n\}$. This yields a set of (auxiliary) link flows $\{y_a^n\}$.

STEP 3. Line search. Find optimal move size α_n that solves:

$$\min \sum_a \int_0^{X_a^n + \alpha(y_a^n - X_a^n)} t_a(w)\, dw$$

subject to $0 \leq \alpha_n \leq 1$.

STEP 4: Move. Set $X_a^{n+1} = X_a^n + \alpha_n (y_a^n - X_a^n)$, $\forall a$.

STEP 5: Convergence test. If a convergence criterion is met, STOP (the current solution is the set of equilibrium link flows); otherwise, set $n = n + 1$ and GO TO STEP 1.

The preceding algorithmic steps are implemented in the computer code as follows. The input of the characteristics of the network, the O-D matrix, link characteristics, and convergence measures, are included in TRAFASN. The initialization STEP 0 takes place in subroutine UE, where all the main steps of the algorithm are controlled. Following the initialization of all the paths to zero flows, subroutine AON is called to initialize the flows on the links to zero. Then the travel times on the links are computed, initially with zero flows. All travel time computations are performed by calling a separate function called COSTFN. Given these travel times, subroutine SHPATH is called, as many times as the number of origins, to identify the shortest path for each O-D pair. Then the flow for each O-D pair is allocated on the links that make up each shortest path. The calculation of the travel times and the allocation of the flows to the links (all-or-nothing assignment) correspond to STEP 1 and STEP 2 of the algorithm, respectively. STEP 3 is controlled by subroutine BISECT, where the move size is determined by a line search using the bisection method. This move size is used in updating the flows (STEP 4), followed by the convergence test (STEP 5), calculated in subroutine UE. The output of the program is controlled by subroutine DUMP.

The two-class user equilibrium problem arises when two classes of users (e.g., cars and trucks) share the use of the physical right-of-way on the highway facilities. The travel times (costs) experienced by one class of users depend not only on the flow of elements belonging to that class but also on the flow of the other class. When the respective effects of the flow of one class on the travel time of the other are not symmetric (e.g., the effect of one additional truck on the cars' average travel time is greater than the effect of an additional car on the trucks' travel time), the resulting user equilibrium problem does not have an equivalent mathematical programming formulation. One of the most commonly used algorithms for its solution is a direct algorithm called the diagonalization algorithm. A discussion of other approaches is given in the review paper by Friesz (*12*).

In the solution of the two-class user equilibrium problem, a separate copy of the physical network is created for each class of users, as described in Mahmassani et al. (*4*). The interactions between classes sharing the same physical links are then represented through the performance (cost) functions associated with each link in the individual network copies. In the general case, these functions would specify the dependence of a link's travel cost on flows on any other link. In the two-class case, the specification of the cost functions reflects the desired dependence between user classes as interactions among links.

At each iteration, the diagonalization algorithm requires the solution of a single-class user equilibrium problem as a subproblem. The latter arises because at the nth iteration, all cross-link effects are fixed at their levels from the $(n - 1)$th iteration, and the cost on any given link is allowed to respond only to its own corresponding flow. This subproblem is solved using the Frank-Wolfe algorithm. Because each iteration of the diagonalization algorithm requires several iterations of the Frank-Wolfe algorithm to solve the diagonalized subproblem, it is more computationally demanding than the single-class algorithm. In addition, because there are as many origin-destination trip matrices as there are classes of users, greater use must be made of the shortest path and the all-or-nothing assignment procedures. Furthermore, the travel cost functions are more complicated, increasing the computational burden for the move size finding.

Nevertheless, the computer code for the diagonalization algorithm, especially for its streamlined versions (*1*,*5*), does not differ significantly from the single-class code. It is composed of the same subroutines, with some modifications to take into account the division of the traffic into trucks and passenger cars. The previously listed subroutines and functions are renamed in this case, in the respective order in which they were previously mentioned, as UETRDIA, UED, AONUED, TRCOST, SHPUED, BISUED, and DUMPUED. For this reason, the modifications performed to vectorize the single-class code are directly beneficial to the diagonlization code. In the next section, these modifications are described for the single-class code, along with computational results with the vectorized code on two networks used in previous numerical experiments (*4*,*5*).

COMPUTATIONAL RESULTS FOR SINGLE-CLASS UE CODE

The purpose of this section is to illustrate the process followed to vectorize the network assignment code and to document

the improvements achievable by different types of modifications. Most of the testing accompanying the various individual changes was performed on a medium-sized network with 182 O-D pairs, 128 nodes, and 336 links. A similar network was used extensively in earlier experiments with streamlined versions of the diagonalization algorithm (*4*,*5*). A maximum of 500 iterations of the algorithm were allowed before the code was terminated for any test run with this network. All runs were performed on the CRAY X-MP/24 using two available Fortran compilers: the CFT 1.15 and the CFT77 v2.0. The CFT 1.15 is written in CRAY assembly language, the CFT77 in Pascal. The CFT77 has superior scalar performance and implements array syntax (arrays handled as entities) and automatic arrays (storage allocated at run time). In many cases, it has closer FORTRAN syntax error handling and vectorizes some loops that the CFT 1.15 would not. The CFT 1.15 generates scalar and conditional vector loops and chooses between the two at run time, whereas the CFT77 generates only vector code and computes the vector length at run time.

Following the steps described earlier, the performance of the code was first assessed without the vectorizing capabilities of the CFT compilers, and a time analysis was performed to determine the most computationally intensive elements of the program. The results are shown in Table 1. The total time to execute was 15.679 sec, using the CFT 1.15 compiler and 14.99 sec using the other compiler (with vectorization blocked in both cases). This compares with 79 sec on a CYBER CDC 170/750 mainframe or about five times more than the supercomputer without any vectorization.

Next, the program was executed by removing the prohibition of vectorization. The results, shown in Table 2 for both compilers, indicate that the execution times for some of the routines were reduced considerably, though not uniformly. A total reduction of 28 percent was achieved by the vectorized compilation using the CFT 1.15 compiler, and of 32 percent using the other compiler, without any program modification. The shortest path routine vectorized quite well, exhibiting a reduction of about 60 percent. The reductions for functions COSTFN and FINT were much more modest, however, less than 5 percent, thereby pointing our efforts toward seeking to improve them.

TABLE 1 EXECUTION TIMES AND PERCENTAGE OF TOTAL EFFORT FOR EACH SUBROUTINE WHEN VECTORIZATION IS BLOCKED (MAXBLOCK = 1) IN COMPILER FOR THE SINGLE CLASS UE CODE ON NETWORK 1

	CFT 1.15		CFT77 v 2.0	
SUBROUTINE	EXECUTION TIME (Seconds)	(%)	EXECUTION TIME (Seconds)	(%)
AON	2.195	(14.00)	2.559	(17.07)
BISECT	3.065	(19.55)	2.412	(16.09)
COSTFN	5.928	(37.81)	6.115	(40.79)
DUMP	0.222	(1.41)	0.202	(1.35)
FINT	0.507	(3.24)	0.521	(3.47)
SHPATH	3.330	(21.24)	2.827	(18.86)
TRAFASN	0.051	(0.32)	0.050	(0.33)
UE	0.381	(2.43)	0.304	(2.03)
Total Execution Time	15.679	(100)	14.990	(100)

TABLE 2 EXECUTION TIMES (IN SECONDS) AND PERCENTAGE OF TOTAL EFFORT WITH VECTORIZATION USING BOTH CFT COMPILERS FOR THE SINGLE CLASS UE CODE ON NETWORK 1

	CFT 1.15		CFT77 v 2.0	
SUBROUTINE	EXECUTION TIME (Seconds)	(%)	EXECUTION TIME	(%)
AON	1.430	(12.68)	0.917	(8.99)
BISECT	1.813	(16.08)	1.517	(14.87)
COSTFN	5.694	(50.50)	5.785	(56.72)
DUMP	0.215	(1.91)	0.201	(1.97)
FINT	0.485	(4.30)	0.487	(4.78)
SHPATH	1.391	(12.33)	1.055	(10.34)
TRAFASN	0.050	(0.44)	0.048	(0.47)
UE	0.198	(1.75)	0.191	(1.87)
Total Execution Time	11.275	(100)	10.199	(100)

To achieve such improvements, one needs to eliminate data dependencies that inhibit vectorization, as discussed earlier. One strategy in this case is to include the travel cost functions within the BISECT routine instead of repeatedly calling a separate function (COSTFN). Calling functions or subroutines in a loop may inhibit vectorization. This change led to a reduction of 1.274 sec (or 11.3 percent) using the CFT 1.15 compiler. However, it was suspected that a further data dependency existed in the loop for computing the link performance functions that inhibited vectorization. These functions have the following general form:

$$t_a(X_a) = t_{oa}(1 + \beta[X_a/C_a]^{\gamma}),$$

where t_{oa} is the travel time on link a under free flow conditions, C_a is a parameter generally interpreted as the capacity of link a, and β and γ are link-specific parameters. The data dependency in the manner in which the computation of these functions was originally coded arises from the separate calculations of the parameters $A1$ and $B1$, as shown in Figure 1. The expressions for these parameters were therefore included directly in the travel time equation. The foregoing changes are shown in Figure 1 as an example of the kind of local code modifications that can dramatically improve the vector performance of FORTRAN codes. The execution time summary following these changes is reported in Table 3 for both compilers. There was a dramatic drop in execution time to 5.568 sec (or a 51 percent improvement over the unmodified code) for the CFT 1.15 compiler, and to 4.061 (60 percent reduction) for the other, primarily because of a drop in BISECT, confirming the prior existence of a dependency that had inhibited the vectorization of the loop.

Given the preceding results, similar changes were made wherever the functions COSTFN and FINT were called. A further step was to specify the $1/C(N)$ in the travel cost equations a variable $C1(N)$, calculated early in the program, so that $X/C(N)$ was transformed to $X * C1(N)$, which eliminates the repetitive division. A division is computationally more demanding than a multiplication on the CRAY. The execution time summary after these and other minor changes is shown in Table 4 for both compilers. The total execution times dropped by about 57 percent and 68 percent relative to the unmodified but compiler vectorized code for the CFT 1.15 and CFT v2.0 compilers, respectively, and by about 69 percent and 78 percent relative to the unmodified and noncompiler

original loop in bisect:

```
      DO 30 N=1, NARC
      X = FL(N) + AMD*(NFL(N)-FL(N))
      A1 = ALP (TYP(N))
      B1 = BET(TYP(N))
      CST = COSTFN (L(N), C(N), V(N), X, A1, B1)
30    D = D + CST*(NFL(N) - FL(N))
```

1st Change: Removing the call function COSTFN

```
      DO 30 N=1, NARC
      X = FL(N) + AMD* (NFL(N) -FL(N))
      A1 = ALP (TYP(N))
      B1 = BET(TYP(N))
      CST = L(N)/V(N)
      IF(C(N). NE.0) CST = CST*(1 + A1*(X/C(N))**B1)
30    D = D + CST*(NFL(N) - FL(N))
```

2nd Change: Incorporating expressions for A1 and B1 directly in the cost (CST) calculation

```
      DO 30 N=1, NARC
      X = FL(N) + AMD* (NFL(N) - FL(N))
      CST = L(N)/V(N)* (1+ ALP(TYP(N))*(X/C(N))**BET(TYP(N)))
30    D = D + CST*(NFL(N) - FL(N))
```

FIGURE 1 Changes to subroutine BISECT to eliminate data dependencies.

TABLE 3 EXECUTION TIME SUMMARY FOLLOWING MODIFICATION OF BISECT AS SHOWN IN FIGURE 1, USING BOTH COMPILERS FOR THE SINGLE CLASS UE CODE ON NETWORK 1

	CFT 1.15		CFT77 v 2.0	
SUBROUTINE	EXECUTION TIME (Seconds)	(%)	EXECUTION TIME	(%)
AON	1.433	(25.73)	0.931	(22.93)
BISECT	1.328	(23.85)	0.654	(16.09)
COSTFN	0.477	(8.56)	0.494	(12.15)
DUMP	0.211	(3.79)	0.203	(4.99)
FINT	0.475	(8.53)	0.488	(12.02)
SHPATH	1.411	(25.34)	1.061	(26.12)
TRAFASN	0.050	(0.90)	0.047	(1.17)
UE	0.184	(3.30)	0.184	(4.52)
Total Execution Time	5.568	(100)	4.061	(100)

TABLE 4 EXECUTION TIME SUMMARY FOLLOWING ALL MODIFICATIONS TO THE SINGLE CLASS UE CODE, USING BOTH COMPILERS, FOR NETWORK 1

	CFT 1.15		CFT77 v 2.0	
SUBROUTINE	EXECUTION TIME (Seconds)	(%)	EXECUTION TIME	(%)
AON	1.357	(27.99)	0.847	(25.66)
BISECT	1.313	(27.07)	0.641	(19.43)
DUMP	0.204	(4.21)	0.206	(6.23)
SHPATH	1.392	(28.71)	1.056	(31.99)
TRAFASN	0.050	(1.04)	0.048	(1.47)
UE	0.533	(10.99)	0.505	(15.23)
Total Execution Time	4.849	(100)	3.301	(100)

vectorized case. The ratio of CDC mainframe to vectorized performance thus becomes of the order of 25 times, compared with about 5 times without any vectorization. This highlights the need for and potential of relatively simple local code modifications to take better advantage of supercomputing capabilities. It is of course possible to improve further on the code's performance; however, the point was reached where the marginal improvements due to additional changes did not justify further effort.

Additional tests of the final vectorized code were performed on a large network of 700 nodes and 1,956 links, confirming the magnitude of the improvement achieved by local vectorization relative to the execution of the unmodified code on the supercomputer and to the CDC mainframe.

COMPUTATIONAL RESULTS FOR DIAGONALIZATION CODE

As explained earlier, the diagonalization program for multiple user classes with asymmetric interactions is very similar to the single-class code. Thus the modifications implemented for the former closely parallel those described in the previous section for the latter. These changes primarily affected the computation of the link performance functions, which are more complicated in the case of multiple user classes, and the BISUED subroutine (the equivalent of the BISECT subroutine for the single-class code). Additional details can be found in the report by Mahmassani et al. (*13*).

The performance of the vectorized diagonalization code was tested on a relatively large network, with two classes of vehicles operating on it. The interactions between vehicle classes are represented in the link performance functions, as described by Mahmassani and Mouskos (*4,5*). The network consists of 364 O-D pairs, 1,400 nodes, and 3,912 links. A total of 25 iterations were allowed before the code was terminated for all test runs. For this network, time analyses were performed for (a) original code with no compiler vectorization, (b) original code with compiler vectorization, and (c) modified code with compiler vectorization. The corresponding execution time analyses are summarized in Tables 5, 6, and 7, respectively, for both CFT compilers.

Comparing the results of Tables 5 and 6, compiler vectorization without code modification leads to an improvement from 23 sec to about 13.5 sec (i.e., a 41.5 percent reduction)

TABLE 5 EXECUTION TIME SUMMARY FOR THE UNMODIFIED DIAGONALIZATION CODE WITH VECTORIZATION BLOCKED

	CFT 1.15		CFT77 v 2.0	
SUBROUTINE	EXECUTION TIME (Seconds)	(%)	EXECUTION TIME (Seconds)	(%)
AONED	1.192	(5.17)	0.949	(4.89)
BISUED	5.726	(24.84)	2.795	(14.40)
DUMPUED	0.434	(1.88)	0.257	(1.32)
SHPUED	10.033	(43.52)	8.429	(43.41)
TRCOST	4.608	(19.99)	5.945	(30.41)
UED	0.245	(1.06)	0.239	(1.23)
UETRDIA	0.815	(3.54)	0.802	(4.13)
Total Execution Time	23.053	(100)	19.414	(100)

TABLE 6 EXECUTION TIME SUMMARY FOR THE UNMODIFIED DIAGONALIZATION CODE WITH VECTORIZATION USING BOTH CFT COMPILERS

	CFT 1.15		CFT77 v 2.0	
SUBROUTINE	EXECUTION TIME (Seconds)	(%)	EXECUTION TIME (Seconds)	(%)
AONED	0.661	(4.90)	0.448	(3.45)
BISUED	2.856	(21.17)	1.983	(15.25)
DUMPUED	0.201	(1.49)	0.136	(1.04)
SHPUED	4.389	(32.54)	3.712	(28.56)
TRCOST	4.440	(32.91)	5.865	(45.13)
UED	0.125	(0.93)	0.084	(0.64)
UETRDIA	0.816	(6.05)	0.770	(5.92)
Total Execution Time	13.489	(100)	12.998	(100)

TABLE 7 EXECUTION TIME SUMMARY FOR THE MODIFIED DIAGONALIZATION CODE WITH COMPILER VECTORIZATION

	CFT 1.15		CFT77 v 2.0	
SUBROUTINE	EXECUTION TIME (Seconds)	(%)	EXECUTION TIME (Seconds)	(%)
AONED	0.423	(6.26)	0.299	(5.18)
BISUED	0.861	(12.74)	0.795	(13.76)
DUMPUED	0.217	(3.21)	0.187	(3.24)
SHPUED	4.326	(63.99)	3.647	(63.11)
UED	0.125	(1.85)	0.084	(1.45)
UETRDIA	0.808	(11.95)	0.766	(13.26)
Total Execution Time	6.759	(100)	5.779	(100)

for the CFT 1.15 compiler and a 33 percent reduction for the other compiler. This time is cut by about half after the code is modified, as shown by Table 7, for a total reduction of about 70 percent, corresponding to a nonvectorized to vectorized improvement ratio in excess of 300 percent, for both compilers. As a reference, the code executed in 126 sec on the CDC mainframe, so the vectorized code on the CRAY performed 22 times better than the unmodified code on the mainframe.

CONCLUDING COMMENTS

The results presented in this paper provide an indication of the magnitude of the reductions in execution time of network assignment codes on the CRAY X-MP/24 supercomputer that can be achieved by the vectorization of the codes, and relative to mainframe computers. For both the single-class user equilibrium and the two-class user equilibrium problem with asymmetric interactions, considerable improvement was achieved following local vectorization by limited modifications to the codes: about 80 percent and 70 percent, respectively, over the unvectorized execution. Our experience confirms the effectiveness of the recommendations followed to optimize these two FORTRAN codes, mainly trying to avoid dependencies within the DO LOOPS. Inserting in line the travel cost functions proved very helpful in both cases. The unmodified codes ran about 5 times faster on the CRAY X-MP without compiler vectorization, and between about 7 and 10 times faster with compiler vectorization, than on the CDC mainframe. However, after the modifications, execution on the CRAY was about 22 times faster than on the mainframe. Of course, generalization of these conclusions requires additional experiments on networks with different configurations and sizes. It is expected that the relative improvement due to the modifications would depend on the extent to which the shortest path routine is called in a particular problem.

It is therefore important to realize that off-the-shelf codes for network analysis originally developed to maximize efficiency on mainframes are not likely to run very efficiently on supercomputers with vector processing capabilities. The results given here demonstrate that relatively simple local modifications can have significant impacts on the vector performance of such codes. The generally applicable guidelines followed in our vectorization of these codes are easy to implement and have been shown to be quite effective.

In this study, no attempt was made to go beyond the local level of code vectorization. It is quite possible that additional improvements can be achieved by using more efficient data structures, or different algorithms, for the overall problem or any of its parts, specifically conceived or selected for their potential for efficient vector performance. Interesting challenges lie ahead along those lines as solution procedures are revised and devised to take advantage of increasingly available innovative hardware. For instance, local modifications in the shortest path routine did not yield significant improvements, suggesting that additional reduction may require more global attempts.

Having established the foregoing results, it is important to ask what their implications might be for research and practice. Should researchers and practitioners attempt to perform all assignment runs on supercomputers? The answer is of course that most everyday applications of traffic assignment models, especially of the fixed-demand single-class variety, will and should continue their migration to microcomputers. The capabilities offered by supercomputers mean that one can address very large-scale problems, and afford greater detail in network representation and, more important, greater realism in the underlying assumptions. For instance, problems with multiple user classes and asymmetric interactions are notoriously demanding computationally; supercomputers offer an attractive computing environment in which to solve such problems and not be discouraged from performing sensitivity analyses. In addition, supercomputer capabilities may lead to breakthroughs in two subjects of current interest to researchers and of great potential practical significance: dynamic assignment problems and the network design problem. Both problems give rise to serious computational hurdles that have considerably slowed progress on their substantive aspects and on their solution in practical applications.

The network design problem belongs to the category of *np*-hard problems. A particular variant of practical interest arises in connection with the selection of truck-related improvements, described by Mahmassani et al. (*4*,*14*), that can be stated as follows: Given a network with known O-D matrices for each category of network users and a number of links n, the problem is to propose various improvements to the links so as to improve operating conditions and service levels offered by the network. If k improvement options are available for each link, the problem's combinatorial complexity rises to k^n. Because the calculation of the travel costs associated with a particular combination of improvements requires the application of a traffic assignment procedure (to find either a user equilibrium solution or a system optimum solution), improve-

ments in the execution of traffic assignment codes have important implications for the size of practical network design problems that can be solved. The encouraging results obtained in this study allow some optimism toward vectorizing transportation network design codes, of which the network equilibrium assignment is a component, and attempting their execution on the CRAY. Furthermore, it would be useful to go beyond local code vectorization to consider global restructuring of the code to achieve greater levels of computational efficiency.

ACKNOWLEDGMENTS

Principal funding for the study on which this paper is based came from a grant from Cray Research Inc. Computing resources for this work were provided by the University of Texas System Center for High Performance Computing (CHPC). The assistance and cooperation of CHPC staff in the course of this study are appreciated. In particular, the contribution of Spiros Vellas to the vectorization of the codes is gratefully acknowledged. The single-class network equilibrium assignment code used in this study is a modified version of a code initially provided by Fred Mannering, presently at the University of Washington, who modified the program originally supplied by Stella Dafermos at Brown University. The authors of course are solely responsible for the content of this paper.

REFERENCES

1. Y. Sheffi. *Urban Transportation Networks*. Prentice-Hall, Englewood Cliffs, N.J., 1985.
2. S. C. Dafermos. Relaxation Algorithms for the General Asymmetric Traffic Equilibrium Problem. *Transportation Science*, Vol. 16, No. 2, 1982, pp. 231–240.
3. A. B. Nagurney. Computational Comparison of Algorithms for General Traffic Equilibrium Problems with Fixed and Elastic Demands. *Transportation Research*, Vol. 20B, No. 1, 1986, pp. 78–84.
4. H. S. Mahmassani, K. C. Mouskos, and C. M. Walton. Application and Testing of the Diagonalization Algorithm for the Evaluation of Truck-Related Highway Improvements. In *Transportation Research Record 1120*, TRB, National Research Council, Washington, D.C., 1987, pp. 24–32.
5. H. S. Mahmassani and K. C. Mouskos. Some Numerical Results on the Diagonalization Network Assignment Algorithm with Asymmetric Interactions Between Cars and Trucks. *Transportation Research*, Vol. 22B, 1988, pp. 275–290.
6. B. L. Buzbee and D. H. Sharp. Perspectives on Supercomputing. *Science*, Vol. 227, 1985, pp. 591–597.
7. S. A. Zenios and J. M. Mulvey. Nonlinear Network Programming on Vector Computers: A Study on the CRAY X-MP. *Operations Research*, Vol. 34, No. 5, 1986, pp. 667–682.
8. S. S. Chen. Large-Scale and High-Speed Multiprocessor System for Scientific Applications. *High Speed Computation*. NATO ASI Series F, Vol. 7. Springer-Verlag, Berlin, West Germany, 1983.
9. *CRAY FORTRAN Optimazation and Performance Analysis*. Center for High Performance Computing (UT CHPC) User Services, University of Texas at Austin, 1987.
10. *User Guide*. Chapter 12: Optimizing Your FORTRAN Code. San Diego Supercomputer Center, San Diego, Calif., June 1987.
11. M. J. Beckman, C. B. McGuire, and C. B. Winston. *Studies in the Economics of Transportation*. Yale University Press, New Haven, Conn., 1956.
12. T. L. Friesz. Transportation Network Equilibrium, Designed Aggregation: Key Developments and Research Opportunities. *Transportation Research*, Vol. 19A, No. 5/6, 1985, pp. 413–427.
13. H. S. Mahmassani, R. Jayakrishnan, K. C. Mouskos, and R. Herman. *Network Traffic Simulation and Assignment: Supercomputer Applications*. Research Report CRAY-SIM-1988-F. Center for Transportation Research, University of Texas at Austin, 1988.
14. H. S. Mahmassani, C. M. Walton, K. Mouskos, J. J. Massimi, and I. Levinton. *A Methodology for the Assessment of Truck Lane Needs in the Texas Highway Network*. Research Report 356-3F. Center for Transportation Research, University of Texas at Austin, 1985.