MATRIX MULTIPLICATION: A VERY SIMPLE PROGRAM WITH SOME INTERESTING CONSEQUENCES

Marcin Paprzycki
Department of Mathematics and Computer Science
The University of Texas of the Permian Basin
Odessa, TX 79762

0. INTRODUCTION

Two major areas dominate the current research and teaching in Computer Science (CS). The first is ***hardware design*** with its main goal of building more powerful computers to store and process larger amounts of data in a shorter period of time. It is most frequently associated with the School of Engineering. The second major area is *theoretical CS* which deals with a variety of issues related to algorithm design. It will be typically housed in the School of Liberal Arts in the departments of CS. Thus institutionalized gap between algorithm design and hardware design is bridged by *experimental CS*, which seeks to establish the best ways of implementing theoretically designed software on existing computers and to determine performance characteristics of the existing computers by developing and running a variety of benchmarks.

In [4], the authors argued for the importance and relative easiness of including experimental CS into undergraduate curriculum. They have discussed a student research project of comparing two search algorithms as an example. One of the compared algorithms was based on a brute force approach, whereas the other was more subtle and was expected to lead to an improvement over the former. Theoretical analysis allowed to determine that there may be circumstances in which the old algorithm will still outperform the new one. The series of experiments performed on a variety of computers in two programming languages proved that the results depended (among other things) on: a) computer hardware, b) characteristics of the compiler (e.g. the speed of the count controlled loop), c) level of optimization requested, d) data type used in the implementation (e.g. short vs. long reals), e) programming language used to implement both algorithms. They concluded that due to the large number of factors influencing the performance of *implemented* algorithms, analytical methods for predicting the relative value of the algorithms may be, for the time being, unreliable. The only way to find out the better algorithm is to run the two codes on a given machine in a given programming language and compare the results.

Their comments are in line with what is suggested in [5]. With the advent of new computer architectures (e.g. introduction of hierarchical memory, vector processors and parallel computers), it becomes clear that the analytical tools developed so far have become obsolete awaiting the development of new tools. It is also not clear whether analytical methods will ever be able to catch up with the developments in other fields of CS. Experimental CS has thus an important role to play in establishing ways of comparing algorithms on particular computers as well as of comparing performance of different hardware platforms.

The aim of this paper is to present another possible student research project that can be used (and was used) for a variety of educational purposes in a variety of courses across the CS curriculum. Section 1 outlines the project. Six methods of multiplying matrices and the nature of the differences between them are described. Section 2 presents and discusses the results of experiments performed on a VAX 8200 and a Cray Y-MP computers. The concluding Section 3 shortly reviews the author's experiences with using this project in a classroom and offers some suggestions about its possible uses.

1. SIX VERSIONS OF MATRIX MULTIPLICATION

Material and graphs presented in this section are primarily based on [2]. Matrix multiplication is a very simple process. The following Fortran code represents the way it is done "by hand" (it is assumed that *A*, *B* and *C* are *N*x*N* matrices, *C* = *A***B*, and initially *C* = 0):

```
      do 10 I = 1,N
         do 10 J = 1,N
            do 10 K = 1,N
10             C(I,J) = C(I,J) + A(I,K)*B(K,J)
```

Permuting the order of loops generates six programs (IJK, JIK, KIJ, IKJ, JKI and KJI). It is easy to convince oneself (by working out a small example for instance) that each of them will generate the same final result as well as that each performs the same number of arithmetic operations. The only difference between them is the order in which data is accessed. The IJK and JIK algorithms are based on inner product calculations, the KIJ and IKJ methods — on row-vector updates, and the JKI and KJI programs — on column-vector operations (cf. Appendix I, for more extensive discussion). As will be shown, this can greatly influence the performance of the program on almost any computer architecture.

To show that the access pattern can play an important role let us examine a very simple example. We will assume that: A. the program is running on a 486 PC with cache memory; B. the communication between the cache memory and the processor is substantially faster than the communication between the main memory and the cache memory; C. the program is written in Fortran so that the matrices are stored in a column major order; D. the cache is able to store two columns of each of the three matrices at a time. The two row-oriented versions of matrix multiplication will require frequent swapping of data between the cache and the main memory. When the IKJ (or KIJ) version of the program is running, two columns of matrix *C* and two columns of matrix *B* need to be swapped *every two steps* of the inner (*J*) do-loop. The two column-oriented versions, on the other hand, will require substantially reduced amount of data movement. When the JKI (or KJI) version of the program is executed, two columns of matrix *A* need to be replaced by the next two columns of *A* after finishing *every two runs* through the inner (*I*) do-loop. The communication overhead can thus hamper the performance of the IKJ and KIJ versions of matrix multiplication substantially, whereas it should have only minimal effect on the performance of the KJI and JKI versions. The two inner product oriented versions should exhibit intermediate performance characteristics, since when one of the matrices (*A* or *B*) is accessed by rows, the second (*B* or *A*, respectively) is accessed by columns.

When applied to a programming language like C or Pascal where matrices are stored by rows, the above explanation will lead to exactly opposite conclusions. In cases of Algol 60 and Ada, whose definitions contain no specification of the array implementation, and Algol 68, which allows for both storage methods, the results will depend on particular implementation and/or programmers decisions.

2. EXPERIMENTAL RESULTS

2.1. VAX 8200

The first series of experiments was performed on a **VAX 8200** mainframe with built-in hardware floating point accelerator, running VAX VMS operating system. We implemented all six versions of matrix multiplication in Fortran. All results presented here are averages of multiple runs (the differences between the runs were rather small -- up to 3%). Timings were obtained using system timers (VAX VMS Library **LIB$**). Matrices were generated by a system random number generator (**MTH$RANDOM** routine from the **MTH$** library). We ran our experiments for short and long integers and short and long reals for matrices of sizes $N = 50, \ldots, 350$. Results are summarized in Tables 1 and 2 respectively.

As predicted above, the column-oriented versions outperform both the inner product and the row-oriented versions of the algorithm. The differences between the results within each of these categories are negligible. Times for long integers and short reals are quite similar, which can be explained by the fact that VAX 8200 has a 4-byte processor. Operations on short integers and long reals require some additional manipulations; moreover, the size of the data object constitutes additional overhead for long reals.

We have calculated the ratio R between the number of operations (additions and multiplications) needed to multiply two matrices of size NxN ($2N^3$) and the time to multiply them.[1] Interestingly, R remained constant only for the column-oriented versions of the program. In all other cases, R was decreasing with the increase of matrix size. At the same time, the number of page faults (indicated by the system timing routines) increased. This is a clear indication that with the increase of the problem size the communication overhead starts to play an important role by reducing the sustainable performance.

2.2. Cray Y-MP 8/864

The second series of experiments was performed on a Cray Y-MP 8/864 supercomputer. All six versions of matrix multiplication were coded in Fortran and compiled by Cray's **cft77** optimizing compiler. Matrices were generated by Cray's random number generator (function **ranf**). The program performance was assessed by the **perftrace** utility, which was also used to average multiple runs. Since perftrace measures the MFlop rate more precisely than time, this measure is reported (cf. note 1). As previously, we ran our experiments for four standard elementary data types and for matrix sizes $N = 50, \ldots, 400$. The results are summarized in Tables 3 and 4.

As was to be expected, the column-oriented versions outperform the others. As before, the results can be split into three groups depending on the index of the innermost do-loop. Since only operations on short reals are vectorized on the Cray, the performance of the column-oriented versions is almost twice as good as that of any version of matrix multiplication on the remaining data types. Of the three data types operated on in scalar mode, operations on both types of integers are twice as fast as those on long reals since the latter are approximately twice as long.[2]

There are, however, some important differences in the performance on the two computer architectures. As on the VAX, the performance rate R was either constant or steadily decreasing, this is not the case on the Cray. For all data types a sudden drop in performance can be observed for $N = 400$. To determine the reasons for the performance degeneration we have run another series of experiments for $N = 1215, \ldots, 1249$. The results are summarized in Figure 1.

Matrix Size	Short Integers						Long Integers					
	IJK	JIK	KIJ	IKJ	JKI	KJI	IJK	JIK	KIJ	IKJ	JKI	KJI
50	1.54	1.50	1.80	1.75	1.34	1.34	1.17	1.10	1.42	1.38	0.97	1.01
100	12.51	12.45	14.80	14.90	11.01	11.10	9.68	9.60	12.17	12.34	8.14	8.19
150	44.44	44.49	53.75	54.11	37.97	38.06	35.74	35.49	57.16	57.66	28.23	28.55
200	109.27	108.55	147.53	148.09	90.32	89.76	96.04	95.54	163.90	163.92	67.13	68.24
250	214.21	213.24	357.63	359.01	174.79	175.59	219.49	218.36	316.58	318.25	129.94	131.41
300	420.28	419.56	481.71	482.42	299.75	301.47	383.61	382.79	548.33	548.25	223.28	226.64
350	721.63	720.40	1009.04	1007.66	470.97	473.21	611.25	611.19	873.98	872.33	356.97	360.8

Table 1. VAX 8200 short and long integers; results in seconds.

Matrix Size	Short Reals						Long Reals					
	IJK	JIK	KIJ	IKJ	JKI	KJI	IJK	JIK	KIJ	IKJ	JKI	KJI
50	1.21	1.20	1.41	1.43	1.05	1.07	1.85	1.87	2.11	2.07	1.74	1.77
100	10.21	10.20	12.21	12.35	8.55	8.53	15.61	15.43	20.32	20.18	13.61	13.61
150	37.26	36.95	56.71	57.24	29.69	29.70	60.15	60.01	84.69	83.90	46.11	45.88
200	100.70	98.68	160.86	161.77	80.49	80.24	155.99	155.35	202.58	201.10	111.18	111.61
250	226.35	225.70	314.71	316.18	137.42	137.34	309.25	308.79	399.58	398.41	217.90	217.52
300	397.33	396.05	543.85	544.43	235.41	237.01	533.17	530.41	682.34	679.08	372.00	373.36
350	635.93	633.66	869.10	869.61	377.27	377.71	857.95	851.80	1090.47	1083.83	600.45	602.00

Table 2. VAX 8200; short and long reals; results in seconds.

To explain this strange behavior we need to consider the Cray's hardware in a little more detail.[3] Three important features of Cray-like architectures contribute to its high level of performance: *pipelining* delivers one result per cycle; *overlapping* executes more than one instruction concurrently; *chaining* allows the results of an operation performed in one functional unit to be fed into another functional unit without being stored in the register. For

MATRIX SIZE	SHORT INTEGERS						LONG INTEGERS					
	IJK	JIK	KIJ	IKJ	JKI	KJI	IJK	JIK	KIJ	IKJ	JKI	KJI
50	60.0	60.1	103.4	103.1	105.4	105.9	59.3	59.8	102.2	102.9	104.6	106.0
100	78.0	78.1	111.3	110.8	115.8	115.8	77.9	78.1	110.4	111.0	115.7	114.9
150	96.0	96.2	116.2	116.2	114.7	117.5	96.1	95.9	115.9	115.6	116.9	116.9
200	98.7	98.7	109.4	109.4	118.3	118.2	98.2	98.2	108.7	108.4	118.1	118.1
250	113.5	113.6	127.9	127.7	128.6	128.6	113.4	113.5	127.6	127.5	128.5	128.7
300	117.7	117.8	123.9	123.9	128.6	128.6	117.9	117.9	124.2	123.9	128.5	128.5
350	118.2	118.2	127.5	127.4	128.2	128.1	118.1	118.2	127.1	127.3	128.1	128.1
400	98.2	98.3	81.3	81.5	127.0	127.2	98.6	98.6	81.7	81.6	127.1	127.1

Table 3. Cray Y-MP 8/864; short and long integers; results in MFlops.

MATRIX SIZE	SHORT REALS						LONG REALS					
	IJK	JIK	KIJ	IKJ	JKI	KJI	IJK	JIK	KIJ	IKJ	JKI	KJI
50	64.9	64.6	134.7	133.6	145.9	146.8	43.1	43.1	56.1	55.9	55.9	56.0
100	79.3	79.1	126.3	124.6	171.6	172.3	47.8	47.8	55.5	55.3	56.1	56.2
150	99.9	100.0	166.3	166.6	183.6	184.2	49.9	49.9	56.1	56.1	56.1	56.2
200	100.5	100.7	107.0	107.1	188.5	190.5	50.3	50.3	53.3	53.2	55.8	56.0
250	129.3	128.9	186.3	186.4	207.7	209.1	53.6	53.6	58.4	58.4	58.6	58.8
300	144.8	144.5	134.5	134.2	204.7	204.5	53.8	53.8	57.4	57.2	58.0	58.1
350	139.4	139.2	185.6	185.9	206.7	207.2	54.2	54.2	57.7	57.7	57.9	58.0
400	96.7	96.7	61.3	61.5	211.6	211.5	50.8	50.8	52.5	52.5	57.8	57.8

Table 4. Cray Y-MP 8/864; short and long reals; results in MFlops.

the problem in question, the performance will be enhanced by the availability of pipelining and overlapping, but it may be diminished by the inapplicability of chaining. Moreover, performance can be hampered by memory bank conflicts. In a Cray-like architecture, the memory cycle is substantially longer than the processor cycle. In order to feed in data to the processor fast enough, memory is divided into independent banks between which the data is distributed. A memory bank conflict occurs when consecutive requests are made for the data stored in the same memory bank. Let us assume that we have 4 memory banks and a 4x4

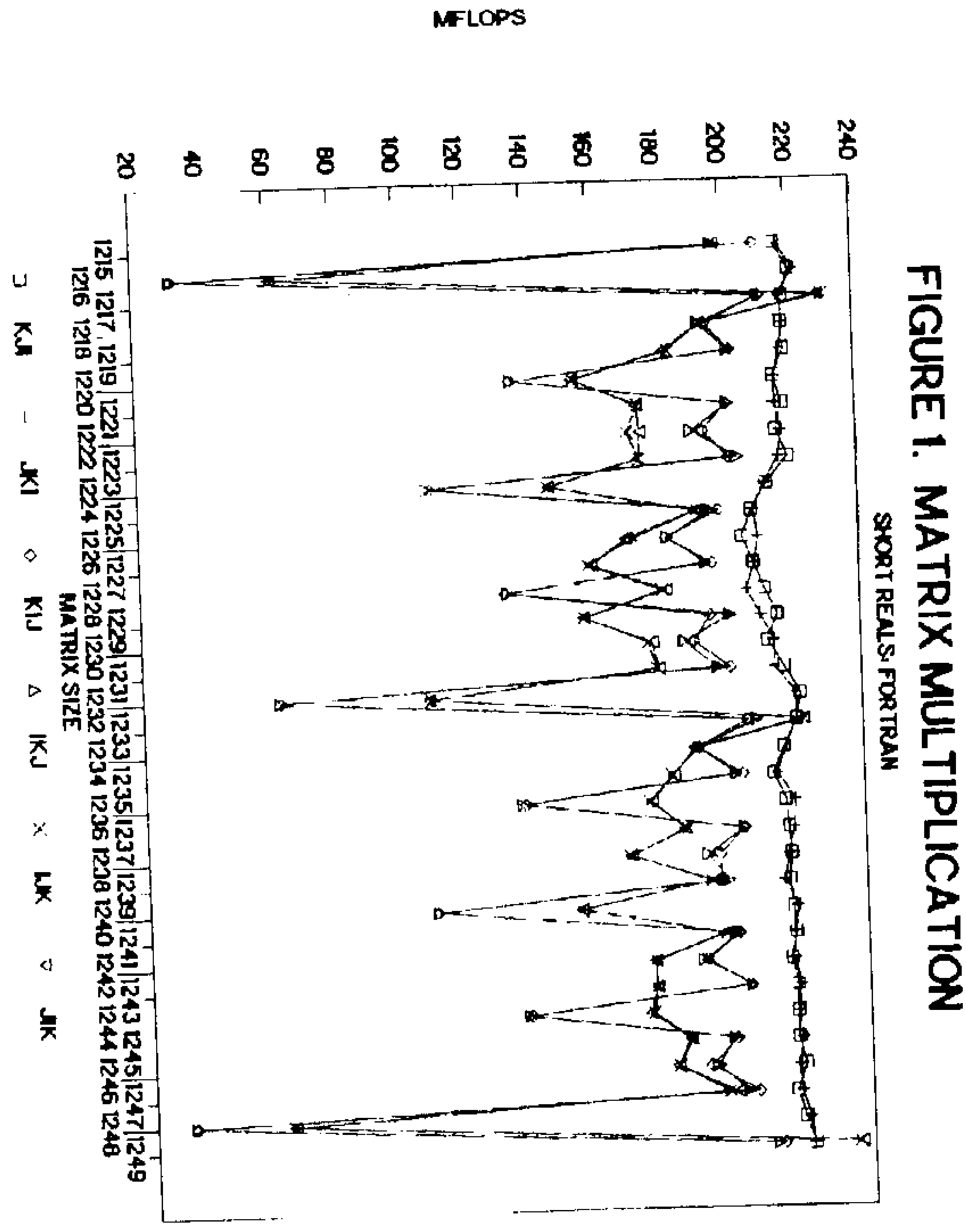


FIGURE 1. MATRIX MULTIPLICATION

matrix is stored in a column oriented fashion.

X(1,1)	X(2,1)	X(3,1)	X(4,1)
X(1,2)	X(2,2)	X(3,2)	X(4,2)
X(1,3)	X(2,3)	X(3,3)	X(4,3)
X(1,4)	X(2,4)	X(3,4)	X(4,4)
I	II	III	IV

If matrix X is accessed column-wise data will be accessed from memory banks I, II, III, IV, I, ... In the case of row-wise access, on the other hand, the data will be sought from the same memory bank creating a memory bank conflict which causes the processor to wait for the data. All matrices of sizes divisible by the number of memory banks (4, in this case) would be susceptible to the same problem.

The inner product based versions of the algorithm are primarily affected by the lack of chaining. At every step, a value of $C(I,J)$ is updated. To assure that every time the most current copy of this element is updated this operation cannot be chained. Since the current $C(I,J)$ needs to be stored in the register before it can be reaccessed by another update, time necessary to perform one store and one load will be "lost" at every step. The IJK and JIK versions are also susceptible to memory bank conflicts as one of the matrices is accessed by rows.

The row-oriented versions of matrix multiplication allow chaining but are highly sensitive to the matrix size (memory bank conflicts). For all odd matrix sizes, IKJ and KIJ outperform the inner product oriented versions. For even matrix sizes, the number of memory bank conflicts generated increases as the matrix size becomes divisible by increasing power of 2. Largest performance decrease occurs for N divisible by 32.[4]

It is also worth mentioning that for large matrices the MFlop rate obtained oscillates around 230-240 MFlops. At the same time, the theoretical peak performance of a one processor Cray Y-MP, as reported by the manufacturer, is 333 MFlops. Given the fact that column-oriented matrix multiplication operates on very long vectors and uses chaining extensively (thus satisfying Cray's optimal performance conditions), it can be stipulated that 240 MFlops (70% of the theoretical peak) is a *practical* peak performance for programs running on a one processor Cray Y-MP [3].

3. CONCLUSIONS

We have presented results of experiments with six versions of matrix multiplication performed on two radically different computers. A project of this kind was used in couple of classes offered at UT Permian Basin. We have assigned it in a Data Structures course (VAX version).[5] It had two goals: to expose students to the consequences of different array implementations and to the effects that different elementary data types used to construct arrays have on the program's performance. In the Programming Languages class, it was used in a comparative study between two (or more) programming languages. In Numerical Linear Algebra, it was used to break the ice between students and the Unix-based Cray (most of our seniors are VMS literate but are rarely exposed to Unix). It also gave the students an interesting background in Cray's hardware. The project can be used in other courses as well. It is appropriate for a Digital Computer Organization course, for instance.

We believe that a project of this nature has an important educational value. Since there are no predetermined answers students find it interesting and challenging. As it is more than simple to implement, students can concentrate on analyzing the results, which is typically one of their weak points. At the same time, the hardware requirements are not extremely high since it can be carried out on any hierarchical memory computer, which will soon become the standard even for PC's. It is enough to have a 486 based PC and a compiler capable of taking advantage of the cache memory to run the simplest version of this project.

We hope that this paper will be able to convince some of the readers that experimental computer science is worth including into the curriculum. It can be a valuable experience and fun for both students and teachers.

NOTES:

1. Divided by 10^6, R approximates what is defined as the MFlop rate of the processor. For single precision reals, the VAX 8200 processor was running at approximately 0.2 MFlops.

2. It should be noted that Cray's arithmetic does not follow IEEE standards. Single precision real provides about 14 digits, and double precision real about 28 digits of accuracy.

3. The discussion of Cray's hardware is based on [1] and [2] which are recommended as source of more information.

4. This is not to suggest that Cray Y-MP has 32 memory banks (For more detailed discussion, cf. [1,2]).

5. A word of caution is in place. As is evident from Tables 1 and 2, matrix multiplication is rather time consuming. When this project was assigned for the first time to a class of 25 with a special credit of finding three largest square matrices that can be multiplied on our VAX, the machine was running 100% CPU load for about two months!

REFERENCES

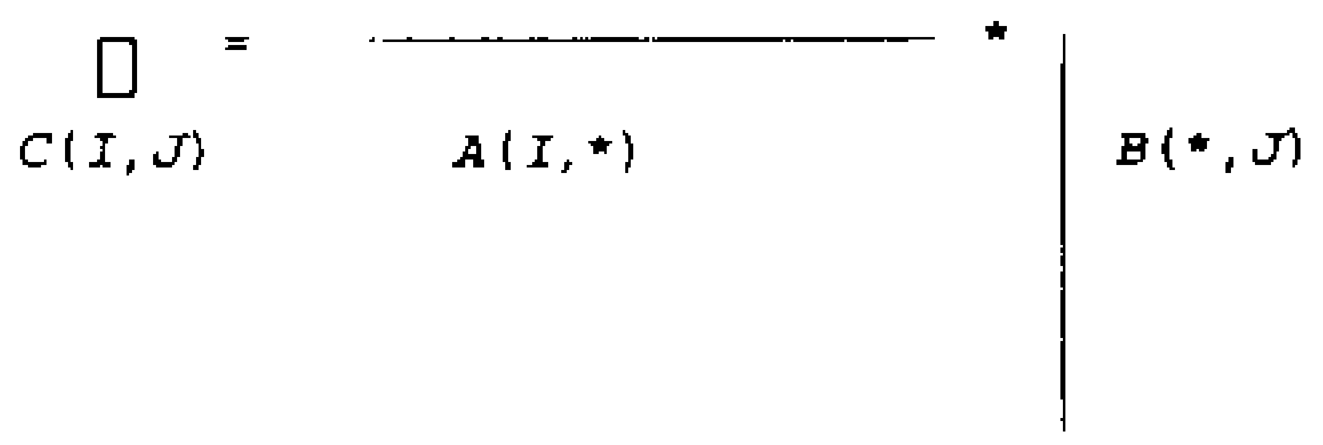
1. Dongarra, J.J., Duff, I.S., Sorensen, D.C., and van der Vorst, H., *Solving Linear Systems on Vector and Shared Memory Computers*, (Philadelphia: SIAM, 1991).
2. Dongarra, J.J., Gustavson, F.G., and Karp, A., "Implementing Linear Algebra Algorithms for Dense Matrices on a Vector Pipeline Machine," *SIAM Review* **26** (1984), 91-112.
3. Paprzycki, M., Cyphers, C., "Multiplying Matrices on the Cray — Practical Considerations," *CHPC Newsletter* **6** (1991), 77-82.
4. Paprzycki, M., Khosraviyani, F., Wagaman, M., "Binary Search on a Linked List — a Research Project in Experimental Computer Science," *Proceedings of The Third Annual South Central Small College Computing Conference* (1992), 16-21.
5. Rice, J.R., *Mathematical Aspects of Scientific Software*, (New York: Springer-Verlag, 1988).

APPENDIX I.

Memory access patterns for matrix multiplication

I. *Inner product* based versions (IJK, JIK)

If we assume that *I* and *J* remain constant, the inner (*K*) do-loop performs the following operation:

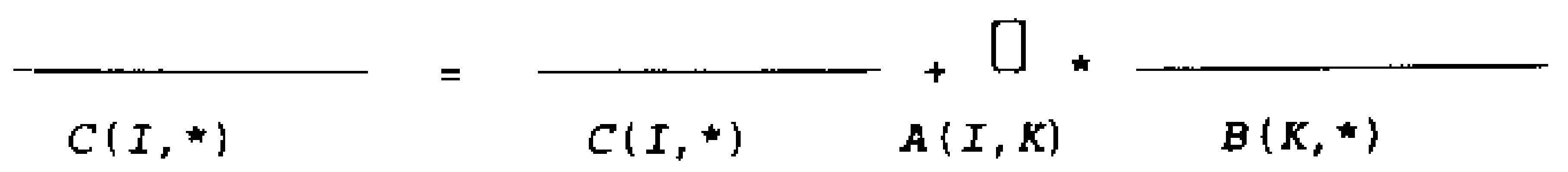


In IJK, matrix *C* is calculated in a row by row fashion where each row is obtained from a sequence of inner products of one row of *A* with all columns of *B*. JIK generates *C* column by column from a sequences of inner products of all rows of *A* with one column of *B*.

II. *Row-oriented* versions (IKJ, KIJ)

The remaining four versions of our program are based on a *vector update* of the form $x \leftarrow x + cy$.

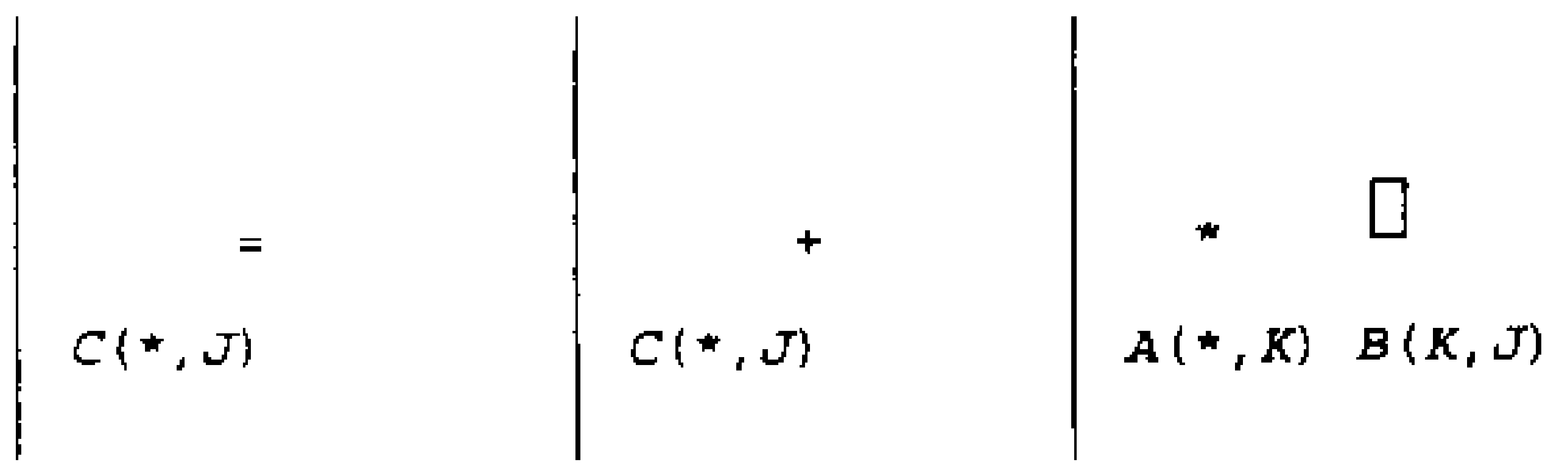
If *I* and *K* are constant the operation performed by the *J* loop can be represented as:



In KIJ, all rows of *C* are updated by a row of *B* scaled by elements of a column of *A*. In IKJ, one row of *C* is calculated at a time by accumulating in it rows of *B* scaled by elements of a row of *A*.

III. *Column-oriented* versions (JKI, KJI)

If *J* and *K* are constants the inner (*I*) do-loop performs the following operation:



In KJI, all columns of *C* are updated by a column of *A* scaled by the elements of one row of *B*. In JKI, a column of *C* is calculated at a time by accumulating in it columns of *A* scaled by elements of a column of *B*.