MATMUL: An Interactive Matrix Multiplication Benchmark

https://people.sc.fsu.edu/~jburkardt/presentations/matmul_1995_psc.pdf

.....

John Burkardt and Paul Puglielli
Pittsburgh Supercomputing Center
Pittsburgh, Pennsylvania, 15213

10 July 1995

Contents

Abstract

This technical report describes an interactive program called **MATMUL**. The **MATMUL** program can be used to make a variety of simple benchmark comparisons involving matrix multiplication. In particular, the user can easily vary the size of the matrix, the leading storage dimension, and the algorithm employed. The program reports the performance of each algorith in a table.

1 Introduction

The Cray can seem like a wonderful, even magical machine. We're told, for instance, that it can add two numbers faster than light can travel six feet. Talk like that is fine for the sales brochure, but it's not a sensible way to think about computers!

Perhaps the idea is right, and it's simply the units of measurement that are inappropriate. Adding two numbers is too small a task, and the distance light travels is too whimsical a measurement.

In this paper, we will try to get to know a computer by doing simple "experiments" on it. We aren't primarily interested in getting a performance rating for computers. We do want to find and explore any unusual behavior we come across.

We've developed a program called MATMUL to use as our exploratory tool. We tried to make the program as flexible as possible, so that it was easy to add

new pieces or change the way the program worked when we found something we hadn't expected.

The most complicated part of the program is the interactive part, which helps the user choose the algorithm and problem size. The simple part of the program carries out the assigned piece of work, and reports the time it took.

Using this approach, we were able to verify some simple facts, such as that a Cray YMP is faster than an IBM PC. (We could also say how much faster, and whether there were some problems the PC could do faster). We were able to see effects we had been trained to expect, such as memory bank conflicts, and the strong influence of index ordering in nested DO loops. But we were also intrigued to stumble across behaviors we didn't know much about, including the power of loop unrolling, the cost of "unusual" arithmetic, and the comparative speeds of C and FORTRAN.

Most of our work was done on the Cray YMP. Some of the behavior we found can be explained only by knowing architectural details of the Cray. But we didn't have to read a Cray manual to find this behavior; we saw it happen to a problem we were interested in.

The program we wrote left many loose ends, but we saw no need to be complete. Our purpose was to do enjoyable "experimental" computer science. We present our methods and results here.

2 The Experimental Program MATMUL

The MATMUL program sets up and solves matrix multiplication problems.

There were many reasons for choosing matrix multiplication:

- It's a simple problem that has a lot in common with the big scientific programs that are usually run on the Cray.
- It's easy to make problems of any size.
- It's easy to compute the amount of work the computer will need to carry out to solve the problem.
- There are a lot of algorithms that have been proposed to carry out the solution.

We did NOT want to use Gaussian elimination as the model problem. First of all, there is already a LINPACK benchmark program in wide use. Secondly, the Gaussian elimination problem is much more involved. It's harder to see the computer's behavior, because the coding is so dense.

We don't really care about the actual answer that is computed by MATMUL. (We do check a single value of the answer, just to make sure the work is getting done!). What we want to know is how long it took THIS computer to solve THIS problem using THIS method.

Even a single timing result from MATMUL is not very interesting. After all, we're probably not sure how long it should take for a particular problem.

But the fun begins when MATMUL is used to make comparisons, when we run MATMUL several different ways. For instance, MATMUL can be used to:

- solve the same problem on different computers;
- compare different methods of solving the problem;
- investigate the cost of higher precision, or complex arithmetic;
- find the "cruising rate" for the computer;
- test programming methods of speeding up an algorithm;
- compare different languages.

Each study can produce unexpected results. But often there is a pattern in these results that suggests an explanation. MATMUL can easily be used to test whether this explanation holds on new problems.

3 How Fast Do Different Computers Solve the Same Problem?

Perhaps the simplest use of MATMUL is as a "stopwatch" program, that allows us to stage a race between different computers.

It's not to hard to get a copy of MATMUL running on different computers. Once we've done that, we simply run MATMUL on each machine, solve the same size problem, and note the time taken. We did this using a simple triple DO loop method, called "IJK", on a small problem of size N=64, and got the following results:

Table 1:

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	513	64	0.003509	524288	149.4199	64.0000	Cray YMP	Fortran
IJK	100	64	0.187488	524288	2.7964	64.0000	DECstation	Fortran
IJK	300	64	0.429000	524288	1.2221	64.0000	VAX/VMS	Fortran
IJK	65	64	266.9004	524288	0.0020	64.0000	IBM PC	Fortran
IJK	65	64	468.5000	524288	0.0011	64.0000	Macintosh	Fortran

There are only two columns of this table that concern us right now: the "Time" and the "Machine" columns. (See Appendix 2 for a complete explanation of the format of these tables). MATMUL reports the number of seconds that it took to solve the chosen problem. We can see that the amount of time required varies by a factor of more than 100,000 between the slowest and the fastest machines!

These results should suggest why the Cray is called a "supercomputer". It really does work faster than others. The rest of our experiments will take place on the Cray, with the VAX/VMS used for comparison. This is mainly to simplify our report; also, we didn't have to wait forever for results!

4 How Much Does Vectorization Help on the Cray?

The Cray is a fast machine. The hardware includes fast processor chips; there is also a software technique called "vectorization" which tries to hurry up the solution of problems involving operations on large lists of data. In FORTRAN programs, this vectorization occurs only for statements inside DO loops, while C programs can achieve vectorization in FOR loops.

Without understanding much about vectorization, we can use MATMUL to investigate how much of the Cray's speed comes from hardware, and how much from software.

We have already seen that if we run the IJK method on a problem of size 64, we achieve a speed of 150 MegaFLOPS. How much of this speed comes from vectorization? Because vectorization is done through software, we can turn it "on" or "off" easily. So to test how much vectorization helps us, we can rerun the same problem with vectorization turned off.

Vectorization can be turned off in a FORTRAN program by inserting the statement

```
CDIR$ NEXTSCALAR
```

just before the DO loop that we want the Cray NOT to vectorize. MATMUL includes just such a loop, and calls the resulting method "SIJK". If we run SIJK on the same problem, we achieve a rate of about 10 MegaFLOPS instead of 170:

Table 2: Comparison of vectorized and scalar IJK loops.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	513	150	0.038936	6750000	173.3608	150.0000	Cray YMP	Fortran
SIJK	513	150	0.719396	6750000	9.3829	150.0000	Cray YMP	Fortran

Remember that a VAX/VMS system seems to solve these problems at a rate of about 1 MegaFLOP. Our new results suggests that the Cray's high speed, as compared with a VAX/VMS, derives in roughly equal parts from its fast processor (10 fold speedup) and from vectorization (15 to 30 fold speedup).

5 What Happens When Problems Get Larger?

A big computer like the Cray attracts users with big problems. Most big problems "grew" out of small ones, and the program that solved the small problem is easily modified to solve the large one. Unfortunately, big problems are different than smaller ones. They're typically much harder. An algorithm that is appropriate to solve a small problem on a microcomputer or even a mainframe, may break down when a big version of the problem is tried on a supercomputer.

The problem, which is common to most scientific problems, is that the hardness of the problem is not a "linear" function of the size of the problem. In plain words, if we try to solve a problem that is twice as large, it takes more than twice the amount of work.

For instance, if we measure the "size" of a sorting problem by N, the length of the list of numbers we are sorting, it is well known that the bubble sort method takes roughly N*N/2 steps to sort the list. That means that a list of 200 numbers is 4 times harder to sort than a list of 100 numbers. This is a "quadratic" increase in difficulty. Solving a linear system using Gaussian elimination, or multiplying two square matrices, are both problems which increase "cubically" in difficulty. That is, the number of operations is roughly related to N*N*N, and hence doubling the problem size causes an eight-fold increase in work.

Frequently, there may be a choice of several methods to use to solve a problem. For small problems, a user may prefer the a simple though inefficient method (a bubble sort), rather than a complicated, efficient one (the Quicksort). For small problems, the advantage of efficiency is not obvious. But when the user tries to solve a large problem with the same simple method, a great deal of computer time will be wasted.

So it's important to be able to estimate the behavior of an algorithm for large problems, and to know about other algorithms that may have better performance. We will see that there is at least one way to multiply two matrices (Strassen's algorithm) that is very much more efficient than the standard ways, in this sense.

6 When Does a Computer Reach its Peak Rate?

Whenever a computer solves a problem, there is a certain amount of work that is done that is not directly related to computing the answer. Such work includes the opening of files, the printing of information to the user, the transfer of control to a subroutine, the collection of data from memory, the control of iterations, and so on. This work is sometimes called "overhead".

When MATMUL solves a small problem, the overhead can be a very significant portion of the computation. This can make the computer achieve a much lower MegaFLOP rate than it is capable of. We would expect, however, that for larger problems, the overhead will become relatively less significant, although it will never go away. For larger problems, we would expect to see the computer reach a typical, limiting speed for floating point calculations.

This is somewhat like the behavior of a sports car traveling between two points. When the points are too close (say, 1000 feet), the car spends most of the travel time accelerating, and then decelerating. The average speed is disappointingly low. But on a longer journey, we can expect the car to spend most of the journey traveling at top speed. Therefore, we could estimate that top speed by dividing distance by time. The corresponding estimate for a computer would be made by dividing the amount of arithmetic work by the required time, to get a "computational rate".

Let's look at the computational rates for a few computers. This rate is typically measured in "MegaFLOPS" (millions of floating point operations per second), which is listed in the table as "MFLOPS". First, we will see how the Cray's rate changes as we increase the problem size:

Table 3: IJK on the Cray, for increasing problem size.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	513	1	0.4074E-05	2	0.4909	1.0000	Cray YMP	Fortran
IJK	513	2	0.5682E-05	16	2.8159	2.0000	Cray YMP	Fortran
IJK	513	4	0.1231E-04	128	10.3964	4.0000	Cray YMP	Fortran
IJK	513	8	0.3757E-04	1024	27.2587	8.0000	Cray YMP	Fortran
IJK	513	16	0.1424E-03	8192	57.5265	16.0000	Cray YMP	Fortran
IJK	513	32	0.6377E-03	65536	102.7648	32.0000	Cray YMP	Fortran
IJK	513	64	0.3401E-02	524288	154.1430	64.0000	Cray YMP	Fortran
IJK	513	128	0.2445E-01	4194304	171.5394	128.0000	Cray YMP	Fortran
IJK	513	256	0.1654	33554432	202.8544	256.0000	Cray YMP	Fortran
IJK	513	512	1.232	268435456	217.8186	512.0000	Cray YMP	Fortran

Here, we're only starting to level off at a speed of 200 MegaFLOPS as the problem size reaches 256.

Now we're not talking about the fact that it takes longer to solve bigger problems. That's obvious. What we're saying is that we solve bigger problems more efficiently than smaller ones.

This effect is more pronounced on powerful machines. Their power is wasted on small problems. Compare the above results with the behavior of the VAX/VMS system for the same set of problems.

Table 4: IJK on the VAX/VMS, for increasing problem size.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	300	1	0.001000	2	0.0020	1.0000	VAX/VMS	Fortran
IJK	300	2	0.001000	16	0.0160	2.0000	VAX/VMS	Fortran
IJK	300	4	0.001000	128	0.1280	4.0000	VAX/VMS	Fortran
IJK	300	8	0.001000	1024	1.0240	8.0000	VAX/VMS	Fortran
IJK	300	16	0.001000	8192	8.1920	16.0000	VAX/VMS	Fortran
IJK	300	32	0.050000	65536	1.3107	32.0000	VAX/VMS	Fortran
IJK	300	64	0.450000	524288	1.1651	64.0000	VAX/VMS	Fortran
IJK	300	128	4.020000	4194304	1.0434	128.0000	VAX/VMS	Fortran
IJK	300	256	34.719997	33554432	0.9664	256.000	VAX/VMS	Fortran

The first strange thing is the behavior of the time. It's stuck at 0.001 seconds from N=1 to N=16. Sadly, this is because the VAX timer is not very accurate, and was going to return a timing of 0.0! We've stuck in a line that forces it to return a value of 0.001 in that case. There's also a "goofy" result at N=16, where we get a MegaFLOP rating of 8, which is probably also part of the inaccuracy of the clock.

Once the problem is large enough that the clock returns values greater than 0.001, we can have some confidence in the results. And what we see is that the VAX is actually SLOWING DOWN for large problems!

It turns out that the VAX was not designed to solve big problems well. Its top computational speed may be about 1 MegaFLOP, but other factors, such as accessing elements of the matrices from paged memory, are slowing it down!

7 How does the Cray Solve a Series of Larger Problems?

If we solve a sequence of problems of increasing size, we will notice another interesting feature of the Cray. For instance, here are the timing results (in ten-thousandths of a second) for a sequence of tests:

Table 5: Problem size versus Cray execution time (0.0001 seconds)

N	60	61	62	63	64	65	66	67
Time	30	31	32	34	35	46	48	49

Here, the jump from N=64 to 65 is very noticeable. We can see similar jumps after N=128, 192, 256 and so on. This is actually caused by the way the special "vectorization" feature of the Cray works.

The Cray is processing our problem in batches of 64. Each time the problem size goes up by 64, the Cray has to introduce a "pause" while it pulls in new data. This "pause" is costly. We can see that 20% of the cost of solving a problem of size 65 is spent handling the very last iteration! So there's a reason to be glad that the C90, the next version of the Cray, will use a vector size of 128!

8 Six Simple Ways to Multiply Matrices

Up to now, we've only used one version of the triple DO loop algorithm. But in a sense, most programs to do matrix multiplication are a variation on the following theme on a triple DO loop:

```
A(I,K) = A(I,K) + B(I,J) * C(J,K)
```

There are six ways to order the loops, even though the actual formula they control remains the same. Surprisingly, the order matters a lot. For a problem of size N=150, the Cray MegaFLOP rates were 170 for four of the orderings, but only 80 for two, the orderings which treat matrix multiplication like a dot product.

We haven't really figured out a good reason for these results. We do know that the dot product calculation might not vectorize as well. Each execution of

Table 6: Six basic algorithms, for N = 150, on the Cray.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	513	150	0.038936	6750000	173.3608	150.0000	Cray YMP	Fortran
IKJ	513	150	0.083022	6750000	81.3042	150.0000	Cray YMP	Fortran
JIK	513	150	0.038396	6750000	175.8016	150.0000	Cray YMP	Fortran
JKI	513	150	0.037936	6750000	177.9294	150.0000	Cray YMP	Fortran
KIJ	513	150	0.082910	6750000	81.4137	150.0000	Cray YMP	Fortran
KJI	513	150	0.038279	6750000	176.3374	150.0000	Cray YMP	Fortran

the dot product loop requires all the partial products to be added to the same quantity, rather than to separate quantities.

9 Advanced Algorithms to Multiply Matrices

Using MATMUL, we can examine advanced multiplication methods that use standard software. We should ask a simple question: do we get a significant speedup in return for the cost of calling a subroutine to do our work? After all, the IJK method is simple, short, and reasonably efficient.

The level 1 Basic Linear Algebra Subprograms (BLAS) are vector oriented routines used as building blocks for packages like LINPACK. The authors hoped that optimized versions of these routines would be developed for all computers, and that programs using them would therefore get high performance.

We can start with the IKJ triple DO loop, and replace the innermost loop with a call to the BLAS routine SDOT. MATMUL contains a copy of the BLAS routine SDOT, but calls it "TDOT" instead. There is also an optimized version of SDOT available in the Cray scientific library, SCILIB. Thus, it would make sense to compare the performance of IKJ, TDOT and SDOT:

Table 7: Compare IKJ, TDOT, and SDOT.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IKJ	513	150	0.8367E-01	6750000	80.6726	150.0000	Cray YMP	Fortran
TDOT	513	150	0.1160	6750000	58.2123	150.0000	Cray YMP	Fortran
SDOT	513	150	0.9449E-01	6750000	71.4346	150.0000	Cray YMP	Fortran

The message here might be that the optimization available in SDOT and TDOT is outweighed by the cost of transferring control back and forth between the main routine and the subroutine.

Similar results were discovered for the "equivalent" codes JKI, TAXPYC, and SAXPYC, as well as the group IJK, TAXPYR and SAXPYR. Whether we used our own FORTRAN copies, or the optimized SCILIB copies, programs using the BLAS routines were slower than the straightforward triple loops.

The designers of the BLAS were themselves disappointed with the resulting performance. However, they guessed that methods involving double and triple

loops might benefit from unrolling the outer loops, not the inner one! Hence they augmented the original level 1 BLAS with level 2 (vector-matrix) and level 3 (matrix-matrix) BLAS.

In particular, they wrote a routine SGEMM that carries out matrix multiplication. Since the SGEMM routine in effect contains the entire triple DO loop, the designers had more opportunities for optimization on the Cray. The designers also used special "block-oriented" techniques to try to reduce the number of times that each data item was read from memory.

The improvements to SGEMM are very clear when we test it with MATMUL. The standard SGEMM routine, here called "TGEMM", runs at a good rate, while the SCILIB optimized version of SGEMM runs at 300 MegaFLOPS! There's also a "super-optimized" version ("SGEMMS") that doesn't perform as well on small problems, but which can seem to run at 400 MegaFLOPS or more on larger problems, which is technically impossible! See *Breaking the MegaFLOP barrier*, PSC News, May 1991.

Table 8: Compare TGEMM, SGEMM, SGEMMS.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
TGEMM	513	150	0.043700	6750000	154.4621	150.0000	Cray YMP	Fortran
SGEMM	513	150	0.022272	6750000	303.0741	150.0000	Cray YMP	Fortran
SGEMMS	513	150	0.023564	6750000	286.4551	150.0000	Cray YMP	Fortran

10 Testing Loop Unrolling

MATMUL allows you to examine loop unrolling performed on the simple IJK method. (Loop unrolling is explained in Appendix 3.

DO loops were supposed to make it possible to replace many statements by one. Why would we want to reverse the process? It turns out that in many cases, a "slightly" unrolled loop will execute more quickly than a more natural version of the same loop. The reasons vary, depending on the computer, and on the actual loop being unrolled.

In matrix multiplication, it is possible to unroll any (or all) of the three loops. MATMUL provides three versions of the IJK loop, with each version unrolling just one of the I, J or K loops to a depth of 4. The three versions are named UIJK, IUJK, and IJUK, with the letter "U" in the name placed just before the name of the unrolled index. "IUJK", for instance, means that the middle "J" loop actually reads "do j=1,n,4". To see exactly what's going on, take a look at the source code for these routines in Appendix 5

When we run these three routines on the Cray, it is interesting to note that unrolling either of the two outer loops dramatically improves performance, to 240 MegaFLOPS. But unrolling the inner, vectorized, loop reduces performance to 95 MegaFLOPS. This effect persisted for a wide range of problem sizes.

This suggests that unrolling should be done on outer loops, and that a small

depth of unrolling can make a noticeable improvement, at least when the "body" of the loop was originally just one or two statements.

Table 9: IJK, and three unrolled versions of it.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	513	150	0.038936	6750000	173.3608	150.0000	Cray YMP	Fortran
UIJK	513	150	0.028064	6750000	240.5252	150.0000	Cray YMP	Fortran
IUJK	513	150	0.027154	6750000	248.5824	150.0000	Cray YMP	Fortran
IJUK	513	150	0.071486	6750000	94.4239	150.0000	Cray YMP	Fortran

11 How Much Harder is Complex Arithmetic?

Scientific calculations often involve the use of the complex number system. In that system, a complex number is a pair of real numbers, one of which is called the "real" part of the complex number, and the second of which is called the "imaginary" part. In printed text, a complex number is often written in a form like "7 + 2 i". Here, the pair of real numbers is 7 and 2, and the "i" tells us that 2 is the imaginary part of the number.

Complex numbers can be added and multiplied, in a similar way to real numbers. The rules for multiplication of two complex numbers are:

$$(a + bi) * (c + di) = (a * c - b * d) + (a * d + b * c)i \quad (1)$$

Thus, there are two obvious costs in working with complex numbers:

- Every complex number takes twice as much storage as a real number;
- Multiplying two complex numbers takes 4 multiplications of real numbers, and two additions, for a total of 6 floating point operations.

Thus, we might expect that MATMUL would take 6 times as long to solve a complex problem as a real one. But since the Cray can do two floating point operations on every step, maybe a better guess would be roughly 3 times as long.

MATMUL includes a single complex algorithm, CIJK, which is simply the IJK method using complex arithmetic. Let's compare the speeds of IJK and CIJK for the same size problem:

Table 10: Comparing real and complex IJK, on Vax and Cray.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	300	150	6.720	6750000	1.0045	150.0000	VAX/VMS	Fortran
CIJK	300	150	11.79	6750000	0.5725	150.0000	VAX/VMS	Fortran
IJK	513	150	0.3964E-01	6750000	170.2971	150.0000	Cray YMP	Fortran
CIJK	513	150	0.1386	6750000	48.7096	150.0000	Cray YMP	Fortran

Notice that the VAX/VMS performance does not decrease as much as we might have guessed. On the other hand, the Cray suffered a more significant drop, of more than a factor of 3.

12 What is the Cost of Double Precision Arithmetic?

We strongly urge our users to avoid double precision computations on the Cray.

The Cray's single precision real data type uses a 64 bit word, which provides similar accuracy to double precision on a 32 bit machine. Specifying double precision on the Cray requests 128 bits of storage, and gives very high accuracy. Unfortunately, double precision Cray computations are extremely slow!

To see why, ask MATMUL to run the double precision implementation of the IJK algorithm. The Cray's MegaFLOP rate plummets to 3! This is 50 times slower than a typical IJK run. In fact, a DO loop containing double precision computations does not vectorize. So it behaves at least as slowly as an unvectorized loop. The fact that the unvectorized calculations also have to be twice as precise accounts for the further slowdown from 10 to 3 MegaFLOPS.

Meanwhile, the VAX is able to produce double precision results almost as fast as the single precision.

Table 11: Comparing real (single precision) and double precision IJK, on Vax and Cray.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	300	150	5.719	6750000	1.1803	150.0000	VAX/VMS	Fortran
DIJK	300	150	6.819000	6750000	0.9899	150.0000	VAX/VMS	Fortran
IJK	513	150	0.038936	6750000	173.3608	150.0000	Cray YMP	Fortran
DIJK	513	150	1.993497	6750000	3.3860	150.0000	Cray YMP	Fortran

13 Integer Arithmetic Should be Faster than Real

We can ask MATMUL to carry out the problem using INTEGER arithmetic. This is done by requesting the "NIJK" method. It's natural to assume that this would always be faster. Integer arithmetic is so much simpler than real arithmetic: no decimal points to worry about, no exponents to align. What happens if we solve the same size problem with real and integer arithmetic?

The VAX results are only mildly surprising. The integer calculations are not very much faster at all. Perhaps we can't very well judge what's harder for a computer.

The Cray results are harder to explain. It seems to be HARDER for the Cray to deal with integers than with real numbers. Note that the Cray has

Table 12: Comparing real and integer (and 46 bit integer) IJK, on Vax and Cray.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	300	150	7.500000	6750000	0.9000	150.0000	VAX/VMS	Fortran
NIJK	300	150	6.600002	6750000	1.0227	150.0000	VAX/VMS	Fortran
IJK	513	150	0.038936	6750000	173.3608	150.0000	Cray YMP	Fortran
NIJK46	513	150	0.090224	6750000	74.8139	150.0000	Cray YMP	Fortran
NIJK	513	150	0.212544	6750000	31.7582	150.0000	Cray YMP	Fortran

two options for storing and operating on integers, full 64 bit integers, or 46 bit integers. Even the 46 bit option (used by NIJK46) is about twice as slow as using real numbers, and the 64 bit option (used by NIJK), is six times slower.

That's very puzzling. What it suggests is that if we had an integer problem to solve, we could simply store it as a real problem, and have the answer faster! This is in fact true. The Cray is not expecting to have to solve integer problems fast. We can actually slow it down by giving it a "simple" problem!

14 Parallel Processing on the Cray

The Cray YMP 8-32 has 8 processors. Normally, the processors operate independently, and do not share tasks or memory. However, the Cray allows a user to insert directives into a program that request that the entire program, or parts of the program, be considered for parallel execution, with portions of the task being assigned to other, cooperating processors.

Fortunately for us, matrix multiplication is ideal for parallel execution. We have implemented a routine "MIJK" which runs the IJK method this way.

We had to make the following changes to do this:

- The program had to be compiled with the compiler option "-Zu". This warns the compiler that parallel processing is being requested. Our compile statement looks like this now:

```
cf77 -Zu matmuluni.f
```

- The triple DO loop in MIJK is preceded by the Cray compiler directive:

```
CMIC$ DO GLOBAL
```

  which offers the loops as a candidate for parallel execution. Actually, the parallel execution will affect the way the I and J loops are handled. For any values of I and J, the inner K loop will be executed completely by just one processor.

- The timing call to SECOND had to be replaced by a call to TIMEF. SECOND computes the elapsed CPU time, whereas TIMEF computes the elapsed wallclock time. MIJK splits the work up among several processors, but the actual amount of work stays the same (or even increases slightly!). The benefit of parallel processing will only be evident in the way that the elapsed wallclock time decreases.

Once we made those changes, we were very pleased to see that we were able to get good speedups for the MIJK algorithm. We ran this program on the Cray during regular production time; we did not reserve the entire machine to ourselves to make these tests.

The maximum MegaFLOP rate we could get would be 2,666, which would occur if all 8 processors were executing at top speed on our problem. We already know that the IJK algorithm, as we have implemented it, does not achieve the top theoretical speed of 333 MegaFLOPS for a single processor, but rather something like 175 MegaFLOPS. So we can even predict a more realistic maximum possible rate for our multitasked version: 8 * 175 = 1,400 MegaFLOPS. And that's assuming we get all 8 processors, which is unlikely during production time.

Here is the result of running IJK once, and MIJK three times in a row. Notice that MIJK seems to have picked up 1, 2 and 8 processors. When you request multitasking on the Cray during production time, you're competing with all the other users. You're sure to get one processor, but you only get more processors if they happen to be idle at the moment your program requests them. And they can come and go during the program's run.

Table 13: Comparing sequential IJK, and 3 runs of multitasked IJK, on the Cray.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	513	150	0.038936	6750000	173.3608	150.0000	Cray YMP	Fortran
MIJK	513	150	0.040757	6750000	165.6168	150.0000	Cray YMP	Fortran
MIJK	513	150	0.026393	6750000	255.7466	150.0000	Cray YMP	Fortran
MIJK	513	150	0.003949	6750000	1709.2017	150.0000	Cray YMP	Fortran

15 Comparing C to FORTRAN

Many of our users prefer C to FORTRAN. But how good is the C compiler at using the Cray's power? In particular, do C programs vectorize well? Can they call SCILIB routines? Can they use multitasking? What are the costs of unusual arithmetics? All of the questions that arose with FORTRAN had to be investigated for C as well.

A version of MATMUL was prepared in C to answer these questions. The results show that the best C code ran just as fast as the best FORTRAN, but

that C versions of some algorithms ran significantly worse than the corresponding FORTRAN. This suggests that C programmers need to be cautious about the structure of nested loops if they want to achieve good performance.

FORTRAN programmers can call a pool of highly optimized routines on the Cray from the SCILIB library. C programmers can also access this or any other FORTRAN library. In order to use FORTRAN libraries, though, a C code must adjust to some special FORTRAN conventions for subroutine calls.

15.1 C calling FORTRAN: Case differences:

The first basic difference is that C allows for case differences in identifiers and FORTRAN does not. This means that in C, a variable called 'x' is different than a variable called 'X', but in FORTRAN they are the same. This not only applies for variables but also for function or subroutine calls. The FORTRAN compiler behaves as though the user's shift-lock key was held down. That means that when a C program calls a FORTRAN routine, it must capitalize the name of that routine.

15.2 C Calling FORTRAN: Passing Variables by Value or Reference:

The next difference we must be aware of is that FORTRAN passes all variables by reference (or address). In other words, any parameter that is passed to a routine can have its value changed and have that change be in effect outside of that routine. Furthermore, the called routine is expecting an address to be passed to it, not a value.

In C all variables are (by default) passed by value (except pointer variables). This means that for every parameter passed to a routine, a new 'local' variable is created inside of the routine which contains the value that was passed. If a change is made to any parameter, that change is not propagated to the original copy of the variable in the calling routine.

So if a FORTRAN routine is expecting an address and C passes a value, you can guess that there will be some confusion. We can however pass a parameter from a C routine to a FORTRAN routine by address, simply by prepending the '&' operator in front of the variable name in the parameter list. The '&' tells the compiler to pass the address of our variable to this routine, not the value, which is what a FORTRAN subroutine or function is expecting.

15.3 C Calling FORTRAN: Passing Constants

This can get tricky when it comes to passing constants. For example in the following FORTRAN call we are passing a constant 15.

```
call some_routine ( a, b, c, 15 )
```

Even though 15 is a constant, SOME_ROUTINE() is expecting an address, not a value. The FORTRAN compiler creates a temporary variable with the value 15 and passes that to the routine.

How do we pass an address of a constant in C? The easiest way is to create a variable and set it to the constant value and pass the address of the new variable, as the following shows:

```
int x=15;
some_routine ( &a, &b, &c, &x );
```

15.4 C Calling FORTRAN: Passing Arrays

The last and most important difference is the way arrays are handled in C and FORTRAN. There are three important differences to be aware of:

First, C starts indices at 0 and goes to N-1, whereas FORTRAN starts at 1 and goes to N. This is a simple difference to cope with.

Secondly, in FORTRAN a two dimensional array is stored as a vector (single dimension array). This is because FORTRAN passes only by reference, so only the start point (first element address) is passed to the routine. We can pass this address in C rather simply. For example if we have the following declaration of 'a' in effect:

```
float a[100][100];
```

Then we can pass the start address of this array as "&a[0][0]".

The last subtle difference between arrays in C and FORTRAN is the way the arrays are stored in memory. Remember that arrays are passed as a vectors in FORTRAN, so if we give the address of the first element, where is the second element? If we have the following 2 dimensional array in FORTRAN:

```
a(1,1)=0.0
a(1,2)=1.0
a(2,1)=2.0
a(2,2)=3.0
```

the address of A(1,1) is passed and A(2,1) is the next value, then A(1,2) followed by A(2,2). Now say we have the same array in C:

```
a[0][0]=0.0
a[0][1]=1.0
a[1][0]=2.0
a[1][1]=3.0
```

We would expect the same as in FORTRAN, but in reality the order is A[0][0], A[0][1], A[1][0] and A[1][1]. So we see that arrays are orientated in columns FORTRAN but by rows in C. This is important when the routine we are calling wants to know the distance between each element. In FORTRAN the distance would be 1, but in C it would be N (where N is the first dimension of the array).

15.5 C Calling FORTRAN: Putting it all together

We used the methods mentioned above and called the SCILIB routines SDOT(), MXMA(), SGEMM(), and SGEMMS() on the Cray. Here is a sample of the results obtained:

Table 14: Comparing SDOT, MXMA, SGEMM and SGEMMS on the Cray.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
SDOT	512	256	0.743565	33554432	45.126446	256.000	Cray YMP	C
MXMA	512	256	0.108822	33554432	308.343700	256.000	Cray YMP	C
SGEMM	512	256	0.110025	33554432	304.970830	256.000	Cray YMP	C
SGEMMS	512	256	0.094922	33554432	353.493474	256.000	Cray YMP	C

Code for calling the SGEMM() routine follows,

```
void TGEMM()
{
  /*
     Declare local variables to use to pass to SGEMMS call instead of
     constant values.  This is done because we must pass the address
     of all parameters to the SGEMMS call.
  */
  int LDA = LENA;
  float alpha = 1.0;
  float beta = 1.0;
  char transa = 'N';
  char transb = 'N';

  TSTART();
  SGEMM ( &transa, &transb, &n, &n, &n, &alpha, &c, &LDA, &b, &LDA, &beta, &a, &LDA );
  TSTOP();
}
```

The code for calling SDOT(), MXMA() and SGEMMS() is in the Appendix 7.

16 Using Pointers to Speed Up Array Access

One of the features of the C language is the fact that anything can be accessed by a pointer. You can access everything from integers to functions with them. However, is it faster to use pointers to access arrays rather than subscript?

We were expecting pointer referencing to be faster than normal subscripting, but we wanted to know how much faster. The speed up would come from the fact that with pointers we would be telling the computer the exact location in memory that we wanted to access, rather than giving the computer a start point and subscripts and telling it to figure the location out for itself. It would seem logical that because we are decreasing the work the computer must do, we would also decrease time.

Two versions of the IJK() algorithm were created to test this theory. They are PIJKA() and PIJKP(). PIJKA() (shown below) accesses the arrays "b" and "c" through pointers, but accesses the array "a" through normal subscripting.

```
void PIJKA()
{
  int i,j,k;
  float *bptr,*cptr,;
  float *bp = \&b[0][0],*cp = \&c[0][0];

  TSTART();
  for ( bptr=bp, i=0; i<n; i++, bptr=bp+LENA)
  {
    for(cptr=cp,j=0;j<n;j++,bptr++,cptr=cp+LENA)
    {
      for ( k=0; k<n; k++, cptr++ )
      {
        a[i][k] = a[i][k] + (*bptr) * (*cptr);
      }
    }
  }
  TSTOP();
}
```

PIJKP() differs from PIJKA() in that it access all three arrays through pointers. Here is the C code for PIJKP():

```
void PIJKP()
{
  int i,j,k;
  float *bptr,*cptr,*bp = &b[0][0],*cp = &c[0][0],*aptr;
  float *ap = &a[0][0];

  TSTART();
```

```
  for ( bptr=bp, i=0; i<n; i++, bptr=bp+LENA, ap+=LENA )
  {
   for ( cptr=cp, j=0;j<n;j++,bptr++,cptr=cp+LENA)
   {
     for ( aptr=ap, k=0; k<n; k++, cptr++, aptr++ )
       *aptr=(*aptr)+(*bptr)*(*cptr);
   }
  }
  TSTOP();
}
```

Here is a sample of the output we received on a DEC Station 5000:

Table 15: Comparing IJK and pointer versions on the DEC Station.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	256	256	14.050943	33554432	2.388056	256.000	DEC Station	C
PIJKA	256	256	11.039225	33554432	3.039564	256.000	DEC Station	C
PIJKP	256	256	12.316536	33554432	2.724340	256.000	DEC Station	C

On this scalar machine, we see better performance with both pointer algorithms; with PIJKA() being the better of the two. This suggests that pointer access of arrays IS faster than ordinary subscripting.

If pointer access is faster, why then is PIJKP() slower than PIJKA()? Looking at the text of the two algorithms (above), we see that by adding the pointer to "a" in PIJKP(), we have also added an extra increment to the very inner loop. Not only is the variable "k" incremented, but so are "cptr" and "aptr". However, in the PIJKA() version, only "k" and "cptr" need to be incremented. By adding the pointer to the "a" array we have added "N*N*N" more pointer increments to our algorithm, thereby slowing it down.

Similar results can be seen on the Cray:

Table 16: Comparing IJK and pointer versions, on the Cray.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	512	256	0.167008	33554432	200.915469	256.000	Cray YMP	C
PIJKA	512	256	0.166260	33554432	201.819438	256.000	Cray YMP	C
PIJKP	512	256	0.207254	33554432	161.900150	256.000	Cray YMP	C

IJK() and PIJKA() give the same performance, but PIJKP() is 40 MFLOPS slower, why? As we said above, we added N*N*N operations. So the overall performance with pointers is at best equal to the performance we could obtain by just using subscripts on the Cray.

It is important to point out that the extra programming difficulty incurred when using pointers, may not net any performance reward; even on scalar machines. The simpler method of loop unrolling (see the routines UIJK(), IUJK()

and IJUK() in Appendix 7) may be a better path to follow when doing numerical work on a computer. Pointer operations are still very useful for string manipulation but the performance increase they yield in numerical work is negligible.

17 Multitasking in C on the Cray

A version of the IJK() algorithm was prepared in C and was multitasked. The result was the MIJK() routine. The text of MIJK() is the same as IJK() except that we had to add one compiler directive, which was:

```
#pragma taskloop defaults
```

This directive was placed before the first (outer most loop) of the routine. This line tells the compiler that the next loop should be multitasked, using the default rules that apply to accessing variables in parrallel.

We also had to change the timing call from cpused() to IRTC(). cpused() returns the total CPU time used, whereas IRTC() returns "wallclock" time. With multitasked code, we are no longer interested in just CPU time, but rather total execution time. Our routine should use the same amouont of CPU time, but spread over up to 8 CPUs simultateously. This will result in total wallclock time decreasing drastically.

You should note however that you will not always get 8 CPU's. You will receive as many that are free when you need them. Even if you get more than 1 CPU during a run, they can be taken away if the system needs them for other purposes. For this reason, a multitasked job run several times, will almost certainly produce a wide range of different times.

Here IJK and MIJK are compared:

Table 17: Comparing IJK and MIJK on the Cray.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	512	512	1.285164	268435456	208.872482	512.000	Cray YMP	C
MIJK	512	512	0.174091	268435456	1541.92517	512.000	Cray YMP	C

(MIJK() received 8 CPU's in the above example.)

18 Conclusions

We hope we have shown you a few things about the Cray. But more importantly, we hope we have shown you a way to think about the Cray, or any other computer. You don't have to believe the manufacturer's brochure, you don't have to know how fast the clock ticks, or how many instructions per second the machine can process. You can judge a computer by its effectiveness on a

problem you choose, and if you pay attention to the results, you can learn more about how to solve the problems you want to solve.

We use these machines to study the world, doing science on computers. We shouldn't hesitate to study computers like we study the rest of the world, doing science TO computers!

A Notes

The MATMUL program was developed at the Pittsburgh Supercomputing Center. Both FORTRAN and C versions are available. The program can be run on many different machines, including the Cray YMP, VAX/VMS, DECstation, IBM PC, and Macintosh. Since the source code is available, you may try to recompile the program for use on other machines as well.

If you have access to the Internet, you can send mail to "remarks@psc.edu" to get a copy of the program, or its documentation, which is in an online document called MATMUL.DOC. Otherwise, send mail to:

```
Consultant
Pittsburgh Supercomputing Center
4400 Fifth Avenue
Pittsburgh, PA, 15213
```

The PSC newsletter article mentioned above ("Breaking the MegaFLOP Barrier") is available online on all PSC systems by typing the "newsletter" command. If you are not a user of the PSC, you can request a copy of the article by sending mail to

```
Documentation Coordinator
Pittsburgh Supercomputing Center
4400 Fifth Avenue
Pittsburgh, PA, 15213
```

B The MATMUL results tables

Throughout this discussion, we have presented the results of using MATMUL in the form of a table. This is, in fact, how MATMUL prints out its results as it is running. And we have simply incorporated those reports into the text.

Each line of the table is the record of one problem solution. The line is labeled with the following headings:

```
 ORDER  LDA   N          Time       Ops    MFLOPS    A(N,N)  Machine  Language
```

and a typical line might read:

```
IJK    513  64      0.003509    524288  149.4199   64.0000 Cray YMP   Fortran
```

The meaning of each of the items is as follows:

ORDER This is the method that the user has chosen, by which to solve the problem. This is done interactively; in this case, the command "ORDER=IJK" had been issued.

LDA The matrices used in MATMUL are stored in two dimensional FORTRAN arrays. These arrays must be at least large enough to hold the matrices, but may be larger. The value of LDA is the "leading" or "first" dimension of the arrays used. On the Cray, there are cases where memory access can be speeded up by a careful choice of LDA.

N This is the size of the problem to be solved. The three matrices, A, B, and C are each set to be N rows by N columns. (They will be stored in LDA by N FORTRAN arrays).

Time This is the amount of time it took to solve the problem, in seconds. On computers that use timesharing, this quantity is the elapsed CPU time. On personal computers, this quantity is simply the elapsed time. When measuring the speed of the MIJK routine on the Cray (a multitasking routine), we report elapsed time, rather than elapsed CPU time.

On some computers, particularly the VAX/VMS, it was not easy to get accurate timings, because the timer routine had limited accuracy (1/60 of a second, for VAX/VMS). This means that timings less than 0.02 may be meaningless, and timings less than 0.10 may be highly inaccurate.

Ops This is the number of floating point operations carried out. It is assumed that multiplying two N by N matrices requires exactly 2*N*N floating point operations. No allowance is made for the fact that the Cray uses 64 bit arithmetic, versus the 32 bit arithmetic used on most other machines. When reporting results for complex, double precision, and integer computations, no allowance is made for the fact that more (or less) work might be required for those arithmetics. And finally, no allowance is made for the fact that some algorithms can compute the result in less than 2*N*N operations!

MFLOPS This is the "speed" at which the computer solved the problem. This speed is measured in "Millions of Floating Point Operations Per Second", and can be computed from "Ops" and "Time" as follows:

$$MFLOPS = (OPS/1,000,000)/Time \tag{2}$$

A(N,N) This quantity is the value of the entry in the N-th row and N-th column of the result matrix. It should always be equal to N, since B and C are both matrices whose entries are all 1.

Machine This records the computer on which MATMUL was run.

Language This records the language that MATMUL was written in. Currently, there are only FORTRAN and C versions to choose from.

C Explanations of terms

Floating Point Operations For scientific computing, it is necessary to estimate the amount of work represented by an algorithm. One way to measure such work is to count the number of additions and multiplications that are to be carried out on real numbers (also called "floating point" numbers). For simple algorithms, this quantity can easily be computed. For instance, in matrix multiplication, every entry of the result matrix is computed by multiplying N pairs of values, and adding each result to a running sum, costing N multiplications and N additions, or 2*N floating point operations. There are N*N such entries to compute, so matrix multiplication will normally cost 2*N*N*N floating point operations.

MegaFLOPS The rate at which a computer carries out a scientific computation is measured in floating point operations per second. Actually, computers are so fast that the measurement is usually made in MILLIONS of floating point operations per second, which is abbreviated as MegaFLOPS or MFLOPS. We have seen that a VAX/VMS system runs at roughly 1 MegaFLOP, a Cray YMP can reach several hundred MegaFLOPS, and simple Macintosh and IBM PC systems run at much lower rates.

Unrolling To unroll a loop means to rewrite it in such a way that several steps of the old loop are carried out in one step of the new loop. Normally, the unrolled loop is "logically" identical to the original loop, in the sense that the same sequence of operations is being carried out, in the same order. Unrolling can also be done on nested loops, in which case the operations may be carried out in a slightly different order.

For instance, the following two loops represent the exact same sequence of operations, but the second loop has been unrolled. Whereas the first loop has a single statement that is repeated 30 times, the second has three statements repeated 10 times. Because the statements in the new loop are carried out three at a time, the loop is said to be unrolled to a "depth" of three.

```
sum = 0.0
do i = 1, 30
  sum = sum + x(i)
end do

sum = 0.0
do i = 1, 30, 3
```

```
    sum = sum + x(i) + x(i+1) + x(i+2)
  end do
```

Vectorization Scientific computing often involves carrying out the same operations on each element of a long list of data. Such a list is often called a "vector". As scientific problems became larger and more time consuming, methods were developed to speed up the solution of such problems on "vectorizing" computers. Through a combination of hardware and software, it became possible to compute certain results much more quickly if the same operations were to be carried out on each item in a vector. Such an operation would usually be represented in a FORTRAN program by a DO loop, whose statements involved addition or multiplication of entries in an array. Matrix multiplication is an example of such an operation.

A vectorizing computer typically has two characteristic speeds: the "scalar" speed, which represents the speed of operation when no vectorizing occurs, and the "vector" speed, when the machine is processing vectors, and operating at a very high rate. Most programs will run at an overall rate that lies between these two values. On the Cray, the scalar speed is roughly 10 MegaFLOPS, and the vector speed is 333 MegaFLOPS.

Memory traffic Once, the reason computers were slow was because their central processors were slow. As faster processors were developed, a new bottleneck was found. It takes a "long" time to fetch data from memory to the processor. Thus, on a given computer, a fast processor will often be sitting idle, waiting for new numbers to be read in from memory.

The movement of data from memory to the processor, and of results back from the processor to memory, is called "memory traffic". Often this traffic represents wasted effort, since a particular number may be moved back and forth between memory and the processor many times.

In some cases, this traffic can be reduced by rewriting the program. For instance, one reason loop unrolling can speed up a program is because it can reduce memory traffic. As a simple example, look at the text of the IUJK routine in the appendix. If you think about it, you should see that each entry A(I,K) is fetched from memory and written back just one fourth as often as in the standard IJK method. Similarly, in the UIJK method, the value C(J,K) must only be read one time from memory to be used in all four of the statements in the inner loop. Thus, again, each entry of C is only read one fourth as often.

Why then isn't the IJUK method on the Cray more efficient also? It turns out that the IJUK method does better on memory traffic, but worse on vectorization, and the net result is a loss in performance.

D The FORTRAN Algorithms

The current FORTRAN version of MATMUL includes a number of algorithms.

Table 18: List of FORTRAN Algorithms.

ORDER	Description
IJK	The DO loop method, with indices I, J, K.
IKJ	The DO loop method, with indices I, K, J.
JIK	The DO loop method, with indices J, I, K.
JKI	The DO loop method, with indices J, K, I.
KIJ	The DO loop method, with indices K, I, J.
KJI	The DO loop method, with indices K, J, I.
CIJK	Complex arithmetic, IJK method.
DIJK	Double precision arithmetic, IJK method.
MIJK	"Multitasked" IJK method, executing in parallel on Cray.
NIJK	Integer arithmetic, IJK method.
NIJK46	Integer arithmetic, IJK method, Cray 46 bit integer option.
SIJK	"Scalar" IJK method. Cray vectorization turned off.
UIJK	IJK loop, with the outer I loop unrolled to a depth of 4.
IUJK	IJK loop, with the middle J loop unrolled to a depth of 4.
IJUK	IJK loop, with the inner K loop unrolled to a depth of 4.
TAXPYC	BLAS calls to TAXPY on columns of matrices.
TAXPYR	BLAS calls to TAXPY on rows of matrices.
TDOT	BLAS dot product routine.
TGEMM	Blas matrix*matrix routine.
MXMA	Cray SCILIB MXMA routine.
SAXPYC	Cray SCILIB copy of SAXPY on columns of matrices.
SAXPYR	Cray SCILIB copy of SAXPY on rows of matrices.
SDOT	Cray SCILIB dot product routine.
SGEMM	Cray SCILIB matrix*matrix routine.
SGEMMS	Cray SCILIB matrix*matrix routine, with Strassen's algorithm.

Here is the FORTRAN source code for the routine that implements the IJK method:

```
subroutine ijk ( a, acheck, b, c, lda, n, ttime )

integer lda
integer n

real a(lda,n)
real acheck
real b(lda,n)
real c(lda,n)
integer i
```

```
integer j
integer k
real time1
real time2
real ttime

do i = 1, n
  do j = 1, n
    a(i,j) = 0.0
    b(i,j) = 1.0
    c(i,j) = 1.0
  end do
end do

call second ( time1 )

do i = 1, n
  do j = 1, n
    do k = 1, n
      a(i,k) = a(i,k) + b(i,j) * c(j,k)
    end do
  end do
end do

call second ( time2 )

ttime = time2 - time1

acheck = a(n,n)

return
end
```

The five other basic DO loop methods differ only in the order of the three loops. Here, for instance, is the heart of the code for "KIJ":

```
subroutine kij ( a, acheck, b, c, lda, n, ttime )

do k = 1, n
  do i = 1, n
    do j = 1, n
      a(i,k) = a(i,k) + b(i,j) * c(j,k)
    end do
  end do
end do
```

The CIJK, DIJK and NIJK algorithms are identical to IJK, except that the matrix variables are declared COMPLEX, DOUBLE PRECISION, or INTEGER, respectively. NIJK46 is the same as NIJK, but is compiled on the Cray in such a way that it uses 46 bit integers.

SIJK is the same as IJK, but is compiled on the Cray in such a way that it is executed without vectorization.

```
      do i = 1, n
        do j = 1, n
CDIR$ NEXTSCALAR
          do k = 1, n
            a(i,k) = a(i,k) + b(i,j) * c(j,k)
          end do
        end do
      end do
```

MIJK uses a Cray directive to run the triple loop using multitasking. The benefit of such a directive depends on the algorithm and the load on the machine.

Except on the Cray, this routine should not be used, and in particular, the call to TIMEF should be commented out.

In order for parallel processing to occur, this routine must be compiled on the Cray with the directive "-Zu"; moreover, the user must set the environment variable NCPUS to the number of processors the user would like. For instance, a C shell user would type:

```
setenv NCPUS 8
```

while a Bourne shell user would type

```
NCPUS = 8
export NCPUS
```

```
cmic$ do global
      do i = 1, n
        do j = 1, n
          do k = 1, n
            a(i,k) = a(i,k) + b(i,j) * c(j,k)
          end do
        end do
      end do
```

IJUK multiplies A=B*C using index order IJK. The K loop is unrolled to a depth of NROLL=4.

```
      khi = ( n / nroll ) * nroll
      do i = 1, n
        do j = 1, n
          do k = 1, khi, nroll
            a(i,k)   = a(i,k)   + b(i,j) * c(j,k)
            a(i,k+1) = a(i,k+1) + b(i,j) * c(j,k+1)
            a(i,k+2) = a(i,k+2) + b(i,j) * c(j,k+2)
            a(i,k+3) = a(i,k+3) + b(i,j) * c(j,k+3)
          end do
        end do
      end do
!
!  Take care of the few cases we missed if N is not a multiple of 4.
!
      do i = 1, n
        do j = 1, n
          do k = khi+1, n
            a(i,k) = a(i,k) + b(i,j) * c(j,k)
          end do
        end do
      end do
```

IUJK multiplies A=B*C using index order IJK. The J loop is unrolled to a depth of NROLL=4.

```
      jhi = ( n / nroll ) * nroll
      do i = 1, n
        do j = 1, jhi, nroll
          do k = 1, n
            a(i,k) = a(i,k) + b(i,j)   * c(j,k)
     &                      + b(i,j+1) * c(j+1,k)
     &                      + b(i,j+2) * c(j+2,k)
     &                      + b(i,j+3) * c(j+3,k)
          end do
        end do
      end do
!
!  Take care of the few cases we missed if N is not a multiple of 4.
!
      do i = 1, n
        do j = jhi+1, n
          do k = 1, n
            a(i,k) = a(i,k) + b(i,j) * c(j,k)
          end do
        end do
```

```
      end do
```

UIJK multiplies A=B*C using index order IJK. The I loop is unrolled to a depth of NROLL=4.

```
      ihi = ( n / nroll ) * nroll
      do i = 1, ihi, nroll
        do j = 1, n
          do k = 1, n
            a(i,k)   = a(i,k)   + b(i,j)   * c(j,k)
            a(i+1,k) = a(i+1,k) + b(i+1,j) * c(j,k)
            a(i+2,k) = a(i+2,k) + b(i+2,j) * c(j,k)
            a(i+3,k) = a(i+3,k) + b(i+3,j) * c(j,k)
          end do
        end do
      end do
!
!  Take care of the few cases we missed if N is not a multiple of 4.
!
      do i = ihi+1, n
        do j = 1, n
          do k = 1, n
            a(i,k) = a(i,k) + b(i,j) * c(j,k)
          end do
        end do
      end do
```

MXMA multiplies A=B*C using the Cray SCILIB routine MXMA.

```
      call mxma ( b, 1, lda, c, 1, lda, a, 1, lda, n, n, n )
```

TAXPYC uses a source code copy TAXPY of the BLAS routine SAXPY to carry out the multiplication columnwise:

```
      do j = 1, n
        do k = 1, n
          call taxpy ( n, c(j,k), b(1,j), 1, a(1,k), 1 )
        end do
      end do
```

TAXPYR uses a source code copy TAXPY of the BLAS routine SAXPY to carry out the multiplication rowwise:

```
do i = 1, n
  do j = 1, n
    call taxpy ( n, b(i,j), c(j,1), lda, a(i,1), lda )
  end do
end do
```

SAXPYC references the vendor's optimized version of the BLAS routine SAXPY to carry out the multiplication columnwise:

```
do j = 1, n
  do k = 1, n
    call saxpy ( n, c(j,k), b(1,j), 1, a(1,k), 1 )
  end do
end do
```

SAXPYR references the vendor's optimized version of the BLAS routine SAXPY to carry out the multiplication rowwise:

```
do i = 1, n
  do j = 1, n
    call saxpy ( n, b(i,j), c(j,1), lda, a(i,1), lda )
  end do
end do
```

TDOT uses a source code copy TDOT of the BLAS routine SDOT to carry out the multiplication:

```
do i = 1, n
  do k = 1, n
    a(i,k) = tdot ( n, b(i,1), lda, c(1,k), 1 )
  end do
end do
```

SDOT references the vendor's optimized version of the BLAS routine SDOT to carry out the multiplication rowwise:

```
do i = 1, n
  do k = 1, n
    a(i,k) = sdot ( n, b(i,1), lda, c(1,k), 1 )
  end do
end do
```

TGEMM uses a source code copy TGEMM of the BLAS3 routine SGEMM.

```
call tgemm ( 'n', 'n', n, n, n, 1.0, b, lda, c, lda, 0.0, a, lda )
```

SGEMM references the vendor's optimized version of the BLAS routine SGEMM to carry out the multiplication:

```
call sgemm ( 'n', 'n', n, n, n, 1.0, b, lda, c, lda, 0.0, a, lda )
```

SGEMMS references the Cray SCILIB version of the BLAS routine SGEMM that has been modified to carry out Strassen's algorithm:

```
call sgemms ( 'n', 'n', n, n, n, 1.0, b, lda, c, lda, 0.0, a, lda, work )
```

E The C algorithms

The current C version of MATMUL includes a subset of the algorithms available in the FORTRAN version.

IJK uses index order IJK.

```
void IJK()
{
  int i,j,k;

  TSTART();
  for (i=0;i < n;i++)
    for (j=0; j < n; j++)
       for (k=0; k < n; k++)
         a[i][k]=a[i][k]+b[i][j]*c[j][k];
  TSTOP();
}
```

For the next "for" loop methods only the kernel of code is shown:

IKJ uses index order IKJ.

```
for (i=0;i < n;i++)
  for (k=0; k < n; k++)
     for (j=0; j < n; j++)
        a[i][k]=a[i][k]+b[i][j]*c[j][k];
```

JIK uses index order JIK.

Table 19: List of C Algorithms.

ORDER	Description
IJK	The for loop method, with indices I, J, K.
IKJ	The for loop method, with indices I, K, J.
JIK	The for loop method, with indices J, I, K.
JKI	The for loop method, with indices J, K, I.
KIJ	The for loop method, with indices K, I, J.
KJI	The for loop method, with indices K, J, I.
DIJK	Double precision arithmetic, IJK method.
MIJK	"Multitasked" IJK method, executing in parallel on Cray.
NIJK	Integer arithmetic, IJK method.
SIJK	"Scalar" IJK method. Cray vectorization turned off.
UIJK	IJK loop, with the outer I loop unrolled to a depth of 4.
IUJK	IJK loop, with the middle J loop unrolled to a depth of 4.
IJUK	IJK loop, with the inner K loop unrolled to a depth of 4.
PIJKA	IJK loop with pointer used to access arrays B and C.
PIJKP	IJK loop with pointers to arrays A, B and C.
TAXPYC	BLAS calls to TAXPY on columns of matrices.
TAXPYR	BLAS calls to TAXPY on rows of matrices.
TDOT	BLAS dot product routine.
TGEMM	Blas matrix*matrix routine.
MXMA	Cray SCILIB MXMA routine.
SAXPYC	Cray SCILIB copy of SAXPY on columns of matrices.
SAXPYR	Cray SCILIB copy of SAXPY on rows of matrices.
SDOT	Cray SCILIB dot product routine.
SGEMM	Cray SCILIB matrix*matrix routine.
SGEMMS	Cray SCILIB matrix*matrix routine, with Strassen's algorithm.

```
for (j=0; j < n; j++)
  for (i=0;i < n;i++)
     for (k=0; k < n; k++)
        a[i][k]=a[i][k]+b[i][j]*c[j][k];
```

JKI uses index order JKI.

```
for (j=0; j < n; j++)
  for (k=0; k <n; k++)
    for (i=0;i < n;i++)
        a[i][k]=a[i][k]+b[i][j]*c[j][k];
```

KIJ uses index order KIJ.

```
for (k=0; k < n; k++)
```

```
    for (i=0; i < n; i++)
      for (j=0; j < n; j++)
          a[i][k]=a[i][k]+b[i][j]*c[j][k];
```

KJI uses index order KJI.

```
   for (k=0; k < n; k++)
     for (j=0; j < n; j++)
       for (i=0;i < n;i++)
           a[i][k]=a[i][k]+b[i][j]*c[j][k];
```

SIJK uses index order IJK with Cray vectorization turned off.

```
#pragma novector
  for (i=0;i < n;i++)
#pragma novector
    for (j=0; j < n; j++)
#pragma novector
       for (k=0; k < n; k++)
         a[i][k]=a[i][k]+b[i][j]*c[j][k];
```

MIJK uses a Cray directive to run the triple loop with multitasking, using IJK order. The benefit of such a directive depends on the algorithm and the load on the machine.

We had to make a new timer call (TMSTART() and TMSTOP()). We needed to get wallclock time instead of CPU time. This is beacuse wallclock time will go down with multitasking, but CPU time should remain the same.

In order for parallel processing to occur, you must set the enviroment variable NCPUS to the number of processors the you would like. For instance, in the C shell you would type:

```
    setenv NCPUS 8
```

while in the Bourne shell you would type

```
    NCPUS=8
    export NCPUS
```

```
#pragma _CRI taskloop defaults
  for (i=0;i < n;i++)
    for (j=0; j < n; j++)
       for (k=0; k < n; k++)
         a[i][k]=a[i][k]+b[i][j]*c[j][k];
```

NIJK multiplies A=B*C using index order IJK with integers

```
  for (i=0;i < n;i++)
    for (j=0; j < n; j++)
       for (k=0; k < n; k++)
         ia[i][k]=ia[i][k]+ib[i][j]*ic[j][k];
```

The entire text of DIJK is shown so that you can see the difference between Cray double precision and double precision on other computers.

DIJK multiplies A=B*C using index order IJK with double precision. A double on the Cray is the same as a float, so we must declare "long double" arrays to try double precision on the Cray.

```
void DIJK()
{
  int i,j,k;
#if (MACHINE == CRAY)
  long double
#else
  double
#endif
    da[LENA][LENA],db[LENA][LENA],dc[LENA][LENA];

  for(i=0;i<n;i++)
  for(j=0;j<n;j++)
  {
     da[i][j]=0.0;
     db[i][j]=1.0;
     dc[i][j]=1.0;
  }
  TSTART();
  for (i=0;i < n;i++)
    for (j=0; j < n; j++)
       for (k=0; k < n; k++)
         da[i][k]=da[i][k]+db[i][j]*dc[j][k];
  TSTOP();
  a[n-1][n-1]=(float)da[n-1][n-1];
}
```

IUJK multiplies A=B*C using index order IJK. The J loop is unrolled to a depth of NROLL = 4

```
  jhi=(n/NROLL)*NROLL;
  jhi--;
```

```
TSTART();
for (i=0;i <n; i++)
  for (j=0; j < jhi;j+=NROLL)
     for (k=0; k < n; k++)
     {
       a[i][k]=a[i][k]+b[i][j]*c[j][k];
       a[i][k]=a[i][k]+b[i][j+1]*c[j+1][k];
       a[i][k]=a[i][k]+b[i][j+2]*c[j+2][k];
       a[i][k]=a[i][k]+b[i][j+3]*c[j+3][k];
     }
for (i=0; i < n; i++)
  for (j=jhi+1;j < n;j++)
     for (k=0; k < n; k++)
       a[i][k]=a[i][k]+b[i][j]*c[j][k];
```

IJUK multiplies A=B*C using index order IJK. The K loop is unrolled to a depth of NROLL = 4

```
for (i=0;i < n; i++)
  for (j=0; j < n; j++)
     for (k=0; k <khi;k+=NROLL)
     {
       a[i][k]=a[i][k]+b[i][j]*c[j][k];
       a[i][k+1]=a[i][k+1]+b[i][j]*c[j][k+1];
       a[i][k+2]=a[i][k+2]+b[i][j]*c[j][k+2];
       a[i][k+3]=a[i][k+3]+b[i][j]*c[j][k+3];
     }

for (i=0; i < n; i++)
  for (j=0; j < n; j++)
     for (k=khi+1;k < n;k++)
       a[i][k]=a[i][k]+b[i][j]*c[j][k];
```

UIJK multiplies A=B*C using index order IJK. The I loop is unrolled to a depth of NROLL = 4

```
for (i=0;i < iroll;i+=NROLL)
  for (j=0; j < n; j++)
     for (k=0; k < n; k++)
     {
       a[i][k]=a[i][k]+b[i][j]*c[j][k];
       a[i+1][k]=a[i+1][k]+b[i+1][j]*c[j][k];
       a[i+2][k]=a[i+2][k]+b[i+2][j]*c[j][k];
       a[i+3][k]=a[i+3][k]+b[i+3][j]*c[j][k];
```

```
      }

  for (i=iroll+1;i < n;i++)
    for (j=0; j < n; j++)
       for (k=0; k < n; k++)
         a[i][k]=a[i][k]+b[i][j]*c[j][k];
```

PIJKA multiplies A=B*C using index order IJK. Arrays b and c are accessed through pointers, but array a is accessed through normal subscripting.

```
  for (bptr=bp,i=0;i<n;i++,bptr=bp+LENA)
   for(cptr=cp,j=0;j<n;j++,bptr++,cptr=cp+LENA)
     for(k=0;k<n;k++,cptr++)
       a[i][k]=a[i][k]+(*bptr)*(*cptr);
```

PIJKP multiplies A=B*C using index order IJK. All the arrays (a,b, and c) are accessed via pointers

```
  for (bptr=bp,i=0;i<n;i++,bptr=bp+LENA,ap+=LENA)
   for(cptr=cp,j=0;j<n;j++,bptr++,cptr=cp+LENA)
     for(aptr=ap,k=0;k<n;k++,cptr++,aptr++)
       *aptr=(*aptr)+(*bptr)*(*cptr);
```

SAXPYC multiplies A=B*C using the vendors copy of the BLAS routine SAXPY. The multiplication is done "columnwise".

```
  for (j=0; j < n; j++)
    for (k=0;k < n;k++)
      SAXPY ( &n, &c[j][k], &b[0][j], &LDA, &a[0][k], &LDA );
```

SAXPYR multiplies A=B*C using the vendors copy of the BLAS routine SAXPY. The multiplication is done "rowwise".

```
  for (i=0;i < n;i++)
    for (j=0; j < n; j++)
      SAXPY ( &n, &b[i][j], &c[j][0], &one, &a[i][0], &one );
```

TAXPYC multiplies A=B*C using a C version of SAXPY. The multiplication is done columnwise.

```
for (j=0; j < n; j++)
  for (k=0;k < n;k++)
    taxpy ( n, &c[j][k], &b[0][j], LDA, &a[0][k], LDA );
```

TAXPYR multiplies A=B*C using a C version of SAXPY. The multiplication is done rowwise.

```
for (i=0;i < n;i++)
  for (j=0; j < n; j++)
    taxpy ( n, &b[i][j], &c[j][0], one, &a[i][0], one );
```

SGEMMS multiplies A=B*C using the Cray SCILIB routine SGEMMS. (BLAS SGEMM modified to use Strassen's algorithm.)

```
SGEMMS ( &transa, &transb, &n, &n, &n, &alpha, &c, &LDA, &b, &LDA, &beta, &a, &LDA, &wor
```

MXMA multiplies A=B*C using the Cray SCILIB routine MXMA.

```
MXMA (  &b, &alpha, &LDA, &c, &beta, &LDA, &a, &beta, &LDA, &n, &n, &n );
```

SGEMM multiplies A=B*C using the vendor's version of the BLAS routine SGEMM.

```
SGEMM ( &transa, &transb, &n, &n, &n, &alpha, &c, &LDA, &b, &LDA, &beta, &a, &LDA );
```

SDOT multiplies A=B*C using the vendor's version of the BLAS routine SDOT.

```
for (i=0;i < n;i++)
  for (j=0; j < n; j++)
    a[i][j] = SDOT ( &n, &b[i][0], &one, &c[0][j], &LDA );
```

TDOT multiplies A=B*C using a C version of the BLAS routine SDOT.

```
for (i=0;i < n;i++)
  for (j=0; j < n; j++)
    a[i][j] = tdot ( n, &b[i][0], one, &c[0][j], LDA );
```

The C version of MATMUL was run on a number of machines, and the timings were recorded in tables.

Table 20: C Timings on the Cray.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	512	256	0.166946	33554432	200.989151	256.000	Cray YMP	C
IJK	512	150	0.037771	6750000	178.708524	150.000	Cray YMP	C
IKJ	512	256	0.702382	33554432	47.772341	256.000	Cray YMP	C
JKI	512	256	1.077998	33554432	31.126614	256.000	Cray YMP	C
UIJK	512	256	0.127904	33554432	262.340194	256.000	Cray YMP	C
PIJKA	512	256	0.175116	33554432	191.612777	256.000	Cray YMP	C
PIJKP	512	256	0.210091	33554432	159.713780	256.000	Cray YMP	C
DIJK	512	256	9.487608	33554432	3.536659	256.000	Cray YMP	C
NIJK	512	256	1.009979	33554432	33.222915	256.000	Cray YMP	C
SIJK	512	256	1.823611	33554432	18.399991	256.000	Cray YMP	C
TAXPYC	512	256	1.203183	33554432	27.888058	256.000	Cray YMP	C
SAXPYC	512	256	2.118891	33554432	15.835850	256.000	Cray YMP	C
TAXPYR	512	256	0.291817	33554432	114.984412	256.000	Cray YMP	C
SAXPYR	512	256	0.212205	33554432	158.122569	256.000	Cray YMP	C
TDOT	512	256	1.445889	33554432	23.206776	256.000	Cray YMP	C
SDOT	512	256	0.690061	33554432	48.625338	256.000	Cray YMP	C
MXMA	512	256	0.108822	33554432	308.343700	256.000	Cray YMP	C
SGEMM	512	256	0.108031	33554432	310.601389	256.000	Cray YMP	C
SGEMMS	512	256	0.094318	33554432	355.757787	256.000	Cray YMP	C
MIJK	512	256	0.022923	33554432	1463.79728	256.000	Cray YMP	C

References

[1] Bailey, Lee, and Simon, *Using Strassen's Algorithm to Accelerate the Solution of Linear Systems*, The Journal of Supercomputing, Volume 4, pages 357-371, 1990

[2] Cray Research, Incorporated *UNICOS Math and Scientific Library Reference Manual*, SR-2081 6.0, January 1991.

[3] Dongarra, Moler, Bunch, Stewart, *LINPACK User's Guide*, SIAM, Philadelphia, 1979 ISBN: 0-89871-172-X

[4] Press, Flannery, Teukolsky, and Vetterling, *Numerical Recipes*, Cambridge University Press, New York, 1986. ISBN: 0-521-30811-9

[5] Strassen, *Gaussian Elimination is Not Optimal*, Numerische Mathematik, Volume 13, pages 354-356, 1969.

[6] Tony Warnock, *Small Steps Toward Great Performance*, Cray Channels, Summer 1988, pages 28-31.

Table 21: C Timings on the IBM RS600.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	256	256	3.670000	33554432	9.142897	256.000	IBM RS6000	C
IJK	256	150	0.770000	6750000	8.766234	150.000	IBM RS6000	C
IKJ	256	256	11.830000	33554432	2.836385	256.000	IBM RS6000	C
JKI	256	256	27.719999	33554432	1.210477	256.000	IBM RS6000	C
UIJK	256	256	3.480000	33554432	9.642078	256.000	IBM RS6000	C
PIJKA	256	256	4.740000	33554432	7.078994	256.000	IBM RS6000	C
PIJKP	256	256	8.880000	33554432	3.778652	256.000	IBM RS6000	C
DIJK	256	256	3.670000	33554432	9.142897	256.000	IBM RS6000	C
NIJK	256	256	6.560000	33554432	5.115005	256.000	IBM RS6000	C

Table 22: C Timings on the HP7000.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	512	256	4.568690	33554432	7.344432	256.000	HP 7000	C
IJK	512	150	0.795392	6750000	8.486382	150.000	HP 7000	C
IKJ	512	256	17.169846	33554432	1.954265	256.000	HP 7000	C
JKI	512	256	30.263432	33554432	1.108745	256.000	HP 7000	C
UIJK	512	256	2.053937	33554432	16.336641	256.000	HP 7000	C
PIJKA	512	256	3.596893	33554432	9.328727	256.000	HP 7000	C
PIJKP	512	256	5.585764	33554432	6.007134	256.000	HP 7000	C
DIJK	512	256	5.600321	33554432	5.991519	256.000	HP 7000	C
NIJK	512	256	5.876466	33554432	5.709968	256.000	HP 7000	C

Table 23: C Timings on the DEC Station.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	256	256	13.980605	33554432	2.400070	256.000	DEC Station	C
IJK	256	150	3.031293	6750000	2.226773	150.000	DEC Station	C
IKJ	256	256	19.914200	33554432	1.684950	256.000	DEC Station	C
JKI	256	256	34.273766	33554432	0.979012	256.000	DEC Station	C
UIJK	256	256	9.832177	33554432	3.412716	256.000	DEC Station	C
PIJKA	256	256	10.914139	33554432	3.074400	256.000	DEC Station	C
PIJKP	256	256	12.511806	33554432	2.681822	256.000	DEC Station	C
DIJK	256	256	19.621279	33554432	1.710104	256.000	DEC Station	C
NIJK	256	256	17.945496	33554432	1.869797	256.000	DEC Station	C

Table 24: C Timings on a VAX/VMS system.

ORDER	LDA	N	Time	Ops	MFLOPS	A(N,N)	Machine	Language
IJK	256	256	28.750000	33554432	1.167111	256.000	VAX/VMS	C
IJK	256	150	5.580000	6750000	1.209677	150.000	VAX/VMS	C
IKJ	256	256	43.119999	33554432	0.778164	256.000	VAX/VMS	C
JKI	256	256	56.930000	33554432	0.589398	256.000	VAX/VMS	C
UIJK	256	256	26.719999	33554432	1.255780	256.000	VAX/VMS	C
PIJKA	256	256	25.879999	33554432	1.296539	256.000	VAX/VMS	C
DIJK	256	256	20.549999	33554432	1.632819	256.000	VAX/VMS	C
NIJK	256	256	13.140000	33554432	2.553610	256.000	VAX/VMS	C