Benchmarking on the Cray YMP

....

John Burkardt

1990

Abstract

This report discusses the idea of benchmarking programs on the Cray YMP at the Pittsburgh Supercomputing Center. It was written around 1990. The machines and programs discussed here are long gone, but the material is not merely of nostalgic interest.

1 Introduction

Benchmarking implies comparison. We might want to compare:

- two programs,

- two versions of a program,

- two compilers,

- two machines.

But remember that you can't compare *one* thing. It's actually hard to know what to make of the fact that your program runs at a rate of 87 MegaFLOPS, or completes in 17 seconds; it's much easier to understand that your program runs twice as fast on an Amiga as it does on a Commodore 64, or that the Pascal version takes 10 times as long to run as the PL/1 version.

After all the theoretical work that goes into writing a program, benchmarking is the practical side. No matter how well the program ought to run, benchmarking reports what really happened. It can be a useful (and sometimes shocking) guide to how well you are doing in matching your programming to what the machine does well.

We can't make benchmarking an end in itself. You're here to get answers, and the only role of benchmarking is to show you whether you're getting those answers quickly and efficiently. Use benchmarking to determine whether there are any serious performance failures in your program. If not, stop benchmarking! Otherwise, use it further to find the worst area of your program, the worst DO-loops, fix those and see if you can move on to the important work.

Having the algorithm right isn't enough! Even as simple an operation as matrix multiplication must be programmed correctly if it is to use the full power of the machine.

2 General Considerations

When timing software, you are usually interested in the elapsed CPU time, and not in the wall clock time. The execution of your job may be interrupted by I/O waits, or by being swapped out. You are not charged for wall clock time, but for CPU.

You want to measure the performance of YOUR program, but benchmarking adds an overhead. You're paying not just for your program, but for the little man who's watching it. His expenses are included in the total, and it's difficult to sort that out. Some benchmarking programs are known to have high overhead, where others are cheap.

The benchmarking tools (with the exception of FORGE) are not commercial tools. Nobody 'wasted' any time making them friendly, easy to use or to understand. They fell off the desk of a programmer at Cray Research, but they're the only thing we have. The documentation is sparse and confusing. The output can be overwhelming.

Benchmarking is easiest to do properly if you have access to the full source code of the program you are interested in. If you are calling many system library routines, or other libraries such as IMSL, you may not be able to carry out many of the tests you might want to make. And if you don't have access to the source code at all, say you are running someone else's executable, the amount of information you can find out is very limited.

Another problem is that real life codes are often enormous. Even if you have source code, many benchmarking programs will churn out reams of output which it is impossible to digest.

Software libraries (BCSLIB, IMSL10, NAG13, SCILIB) are generally well written, correctly implemented, and vectorized where possible. You should prefer them, by default, to your own routines. But the compiler is very good, and library software may be unsuitable for your problem. A simple example: approximate integration. The library routine typically contains a loop:

```
do I = 1, n
  sum = sum + weight * f ( x(i) )
end do
```

where F(X) is a function you write that evaluates the function. On the Cray, the occurrence of the mysterious function F(X) is enough to keep the compiler from trying to vectorize this loop. But if you simply inserted the code for your formula into the loop, the compiler would be able to understand what is going on, and schedule the loop for fast execution:

```
do I = 1, n
```

```
      sum = sum + weight * sin ( x(i) )
   end do
```

3   Special Features of the Cray YMP

I/O is much slower than computations. Unless you have a real interest in I/O measurements, you should time portions of code that *do not* contain READ or WRITE statements.

Multitasking has not been integrated with the benchmarking tools. That means that it is not possible, at the moment, to get a good idea of the performance of a code that uses multitasking, except through wall clock time. We will ignore multitasking in this discussion.

The processor is only so fast. The Cray picks up its speed by vectorization (and multitasking). Things to look for are DO loops that don't vectorize, or don't vectorize well. Also, subroutine calls have an 'overhead', so calling a simple subroutine thousands of times is wasteful. I/O should be done unformatted where possible, and in large 'chunks'.

A naive approach would assume that the Cray can compute 64 results as fast as it can compute one. This is not true! Every vectorized DO loop has a startup and finishup phase that is part of the overhead. The "64-computations-at-a-time" in fact occur in a pipelined manner. This is very fast, but significantly slower than a single computation. In fact, it can be to your advantage to AVOID the vectorization of so-called "short loops".

The Cray works at maximum efficiency when all the features of its processors are hard at work. This means that a 'skimpy' DO-loop will vectorize, but not be as efficient as a more generous loop. With a longer loop, the compiler has more statements to play with, and can orchestrate the fetching and computation better.

As an example of this point, loop unrolling can achieve significant speed ups. The loop:

```
DO I=1,N
  SX(I)=SX(I)+SA*SY(I)
end do
```

is relatively 'skimpy'. The equivalent (if N is even, anyway) loop

```
DO I=1,N,2
  SX(I)=SX(I)+SA*SY(I)
  SX(I+1)=SX(I+1)+SA*SY(I+1)
end do
```

may execute twice as fast!

A better example is:

```
do I=1,N
  X(I)=3.0*Y(I)
```

```
      end do
      do I=1,N
        Z(I)=SQRT(Y(I))
      end do
      do I=1,N
        W(I)=X(I)+Y(I)+Z(I)
      end do
```

which will run much slower than the equivalent:

```
      DO I=1,N
        X(I)=3.0*Y(I)
        Z(I)=SQRT(Y(I))
        W(I)=X(I)+Y(I)+Z(I)
      end do
```

The second form allows the program to fetch a single item of data, Y, perform the calculations, and then write 3 items back to memory. The first form required five separate fetches of data to do the same work. Moreover, the second form is a "richer" DO loop. The processor can do multiplication of 3.0*Y(I) at the same time as the addition can be done!

4   Standard Benchmark Programs

In the case where you want to compare the performance of machines, rather than the performance of software, you need a standard program that you can run on each machine, inserting a few timing calls perhaps, so that you can compare speed, or the ability of the compiler to vectorize loops or execute instructions in parallel. A few such programs are available at the PSC. Source code, documentation and other material is available to you. See the final section for some details.

Benchmark programs testing computational speed include:

- **EXPORTS** A set of five benchmark programs used to rate computers, to determine which ones may not be shipped overseas because they're too fast.

- **LBENCH** The LINPACK benchmark. A document is available containing the timings and corresponding MFLOP rate for this program on various machines. The Cray runs some versions of this test at roughly 150 MFLOPS.

- **MFLOPS** The Livermore Loops program. This package is so pervasive that computer makers sometimes deliberately optimize their compilers to handle certain of these loops and inflate their ratings. This program runs at an average MFLOP rating of 125 on the Cray.

Table 1: The problem set

Name	Task
SOLVE	Factor and solve 100 dense linear equations.
INVERT	Invert a dense matrix of order 100.
MULT	Multiply two matrices of order 500.
EIGEN	Find the eigenvalues of a matrix of order 100.
ODE-RK	Solve a scalar differential equation by Runge-Kutta method.
ODE-AB	Solve a scalar differential equation by Adams-Bashforth method.
FFT	Compute the fast Fourier Transform of a vector of 4096 real values, then invert.
SORT	Sort a vector of 20,000 real numbers.

- **NASKER** The NASA Ames Kernel. Seven subroutines, carrying out matrix multiplication, Fourier Transforms, Cholesky decomposition, and so on. This program runs at roughly 99 MFLOPS.

- **WHSTONE** The Whetstone benchmark, one of the original attempts at estimating machine performance.

- **PSR** The examples discussed in the Levesque and Williamson book. There are also five large production codes useful for practicing how to benchmark.

- **AUTOTASK** Sample FORTRAN subroutines, and a suite of tests from PSR (Pacific Sierra Research) for demonstrating how to use Autotasking.

- **VECTORIZE** A large set of subroutines to feed the compiler, checking for whether loops are vectorized. VECTOR.FOR is a set of 100 loops prepared by Jack Dongarra. The coming CFT77 3.1 compiler manages to vectorize 83 of the loops (the current record!). VECTOR1, VECTOR2, VECTOR3 and VECTOR4 are similar compilations of DO LOOPS.

5 Benchmarking a Mathematical Library

One of the great software issues at PSC is the dominant role that IMSL version 9.2 plays in our usage. Competing software from other vendors is available, which, it is claimed, is highly vectorized for the Cray. Moreover, IMSL responded to user complaints by producing version 10.0, which they attempted to vectorize. However, because of the law of inertia, users persist in using version 9.2. Perhaps we haven't made the case strongly enough, or perhaps users, like an enormous ocean liner, just take a terribly long time to change course.

I tried to measure the differences between the various software packages by using a standard set of problems:

I wrote calling programs that defined each of these problems, and called the appropriate routine from each of our common libraries for a solution. I timed each process and made a table for comparison.

Table 2: Times in milliseconds, under the old COS operating system:

Name	IMSL9	IMSL10	NAG11	NAG12	BCSLIB	VECTPAK	SCILIB	SLATEC
SOLVE:	66	16	19	18	8	—	14	—
INVERT:	159	40	45	67	34	—	39	—
MULT:	4091	1407	1949	1957	1291	501	1275	—
EIGEN:	1419	1099	219	254	210	—	210	—
ODE-RK:	11	54	17	21	9	—	—	11
ODE-AB:	38	72	65	72	34	—	—	46
FFT:	48	31	22	9	8	5	4	4
SORT:	166	116	176	185	59	58	95	156

Now here we finally see some clear evidence that IMSL 10.0 does better than IMSL 9.2 on linear algebra problems. Further, BCSLIB seems to have very strong performance down the line.

6 Benchmarking Utilities Supplied by Cray

When benchmarking, we are interested in determining whether our code runs well enough that we can leave it alone, or identifying problem areas that we should try to fix.

The first thing benchmarking tools can do for us is tell us that there are problems. The crudest measure of performance is the total program MegaFLOP rating. On the YMP, a rating below 20 MFLOPS represents scalar code. A rating of 100 is superior, and 300 is extraordinary.

Another thing that tools can do is identify subroutines that carry out the greatest amount of work, and tell us whether the subroutine is called many times or once.

Finally, we would like information on actual DO loops that do not vectorize, or that are executed the most, or use the most time.

Cray Research supplies some built in benchmarking tools, available in many different forms. A brief summary includes:

- **FLOWTRACE** Lists total time spent in each subroutine. Must have access to the source code. Can be used with C, FORTRAN or PASCAL. Has *some* overhead.

- **FORGE** Proprietary "expert system" program. Helps you analyze the performance of the code, and helps you improve it. Much easier to use if you have a workstation. Takes some getting used to. This program requires extensive interaction with the user. It is not a "fire and forget" utility. Requires source code access. Can be used with FORTRAN.

- **FTREF** Produces a "static" calling tree, and analyzes common block usage. Requires source code access. Can be used with FORTRAN.

- **HPM** reports on the total MFLOP rating of the program. Does not require source code access. Can be used with programs in any language. Has *no* overhead.

- **RTC** Integer version is called IRTC. Wall clock routines. Return the number of 'ticks' that have elapsed since last called. This includes time you were swapped out. Appropriate for multitasking, but not otherwise. Requires access to source code. Can be used with any language, easiest with FORTRAN. There is *some* overhead.

- **LOOPMARK** Displays which loops were vectorized. (Static analysis) Requires access to source code. Can only be used with FORTRAN.

- **PERFTRACE** reports on the MFLOP rating of each subroutine. It reports the same quantities as HPM, but broken down by subroutine. Hence, can be used to compare subroutines, finding the most heavily used, the most I/O bound, the one that computes the most. Requires access to source code. Can be used with C, FORTRAN, or PASCAL. Has *considerable* overhead.

- **PROF** reports the activity of portions of code. PROF divides the program into little pieces. Usually these pieces are smaller than subroutines, but bigger then single lines of code. During program execution, it checks which little piece is actually executing at each moment. If your subroutines are 'small', then PROF will not analyze them any further, unless you change the default size (4 words) of the typical piece of code that PROF monitors. Must have access to the source code. Source may be in any language. There is *no* overhead.

- **SCOUNT** Lists how many times each line was executed. Useful for finding dead code, heavily used loops, determining whether a particular IF condition is 'usually' true or false. Programming bugs and design flaws can show up with this program. Requires source code access. Can be used with FORTRAN. There is *some* overhead.

- **SECOND** Elapsed CPU time routine. Returns CPU elapsed since last call. Tedious to use for large programs, but is very flexible. Requires source code access. Can be used with any language, but easiest with FORTRAN. There is *some* overhead.

- **TIMEF** Returns wall clock time in milliseconds. Requires source code access. Can be used with any language, but easiest with FORTRAN. There is *some* overhead.

7 A Sample Program

A single program was passed through the benchmarking programs. This program comprises a driver program that sets up a matrix and right hand side,

Table 3: CPU time as reported by SECOND:

Routine	Seconds
TGEFA	0.221
SGEFA	0.551
TGESL	0.006
SGESL	0.002

and the LINPACK routines required to solve this linear system. We are using source code for the LINPACK routines, but we have renamed each routine to avoid the possibility of confusion with the SCILIB versions. We solve a system of 500 equations.

The individual routines that make up this program are

- **TGEFA** factor the matrix A.

- **TGESL** solve A*x=b for x, given b and a factored matrix A,

- **ITAMAX** return the index of the largest entry of a vector x.

- **TAXPY** compute y=sa*x+y, for a scalar sa, and vectors x and y.

- **TSCAL** compute x=sa*x, for scalar sa, and vector x.

Theoretical calculations tell us that most of the time should be spent in TAXPY.

8 Benchmarking the Sample Program

Note that these routines (under slightly different names) are part of SCILIB. SCILIB routines, in most cases, have been optimized and written in CAL, the Cray Assembly Language. Thus, I can make a comparison run (always a good idea when benchmarking) by calling the SCILIB routines instead. Naturally, we expect to get some speedup by going to SCILIB. But in fact, we slow down! Our first insight: something is wrong with SCILIB, or the CFT77 compiler is better than it should be.

Now let's just record the overhead from benchmarking by checking the run times for these routines when run with the various benchmarkers. These timing results are for a full matrix of order 500 by 500.

Notice that only HPM, PROF and SCOUNT manage to keep the overhead low. At least for this problem, the other programs have enormous overhead. This overhead makes the reports from these programs somewhat dubious.

Table 4: CPU time when "tool" is included, as reported by SECOND:

Tool	TGEFA	TGESL
HPM	0.221	0.006
FLOWTRACE	2.792	0.026
PERFTRACE	8.185	0.069
PROF	0.222	0.006
SCOUNT	0.261	0.008

9 What Do the Results Mean?

9.1 HPM output

About the only useful number from HPM is the overall MegaFLOP rating, which HPM reports as 4.36. This is a rather disappointing number, since the Cray can, theoretically, run at a peak of about 250 MegaFLOPS. Can we find some excuse? This rating is based on the the total execution time, including the HPM overhead. This is averaged over the whole program, including the time it takes to print messages to the log file, and to assign values to the matrix. The HPM output is of limited use for this program.

9.2 PERFTRACE output

PERFTRACE produces an analysis of each routine in the package. Although it has a significant overhead, it does produce a report which is a little more easy to absorb. We see that TSCAL is the best vectorized at 41 MFLOPS. (We ignore PERFPRB itself, listed as running at 107 MFLOPS, since there is no significant computation there.) ITAMAX is the worst performance, but this is to be expected. Searching for the maximum entry in a vector is not vectorizable. Looking at the amount of work done (as opposed to the speed), we can tell that TGEFA and TAXPY are working the hardest.

9.3 PROF output

The first thing to notice about PROF is the low overhead. The program runs almost as fast as with no benchmarking. This suggests that the results of PROF will not be distorted. However, since PROF is a statistical program, it is important to make sure that enough hits are made to represent a good sample. PROF again reports usage of TAXPY and TGEFA. About 67% of the execution time was spent there. Since the figures PROF reports add up to 81%, the actual percentage is probably more like 84% (.67/.81). Also, note that although PROF can analyze pieces smaller than a subroutine, our subroutines are too small to get any deeper attention by PROF.

9.4 SCOUNT output

Although this program is very unpretentious, the output it gives is quite useful and understandable. The DO 30 loop in TGEFA is very busy, as are some loops in TAXPY and TDOT. Note also that we can see large regions of "dead" code in the routines. Note in particular that we do not pivot in TGEFA. This example, by accident, never has to pivot. We might be able to cause one or two loops to vectorize by removing checks we don't need.

9.5 FLOWTRACE output

Note that FLOWTRACE points out that TAXPY is called 125,749 times. Note that calls to a subroutine take time. Maybe we should consider putting that code "in-line"? FLOWTRACE also reports that TAXPY and TGEFA use the most time, and also cannot get percentages that add up to 100%.

9.6 LOOPMARK output

LOOPMARK gives us a compiled listing with vectorized loops marked. We see that the assignment of A and B does not completely vectorize...It can't, since it's a double loop. We note that TGEFA and TGESL have loops in them, but don't vectorize. The compiler tells us one reason, but we could easily rewrite that problem out. The real problem is that these loops are really "outer" loops. Each of these loops sets up some information, and calls TAXPY, which contains a loop that does vectorize,

9.7 FTREF output

The output of FTREF is not very exciting for this program. It's too small, and there aren't any COMMON blocks. We get a dynamic calling tree from FLOWTRACE, which gives us more information. So I've included output of FTREF for a different program, which does have COMMON blocks.

10 What Can We Do?

The first thing we should do is minimize further work. SCOUNT has shown us code that is *never* accessed. We can delete it all (assuming this is the *only* matrix we will ever want to solve, of course!). Moreover, SCOUNT shows us that for this matrix, we never need to pivot. That means we can get rid of the code that searches for the pivot (ITAMAX, which LOOPMARK shows us doesn't vectorize), the code that moves entries around, and the array IPVT that is used.

FLOWTRACE shows us that TAXPY is called a "grotesque" number of times. Subroutine calls are an overhead that can be gotten rid of. In fact, the actual operation of TAXPY is very simple for this code: add a multiple of one row to another. This is a do loop with one line.

Table 5: Results of improved code - worse performance!

Routine	Seconds
TGEFA	0.556
TGESL	0.003

Table 6: Results of improved code, with TAXPY restored

Routine	Seconds
TGEFA	0.222
TGESL	0.003

Finally, we try to avoid memory bank conflicts by setting LDA to N+1. In fact, because the LINPACK routines are "column oriented" we don't gain much by this at all.

Now if we repeat the benchmarking tests on this code, the results are about the same as for SCILIB (that is, we have *lost* our advantage!).

Well, after all that work, that's disappointing! What could be going wrong? Notice how complicated the TAXPY routine is? In particular, notice the "unrolling" of the DO loops. Could this be involved? Only one way to check it. Let's leave all our other improvements in and restore the call to TAXPY. And lo, we have back our old timings.

To prove that TAXPY was actually helping us, let's unroll the loops even further, to a depth of 8, and rerun the code:

Well, after all that work, the moral of this story might be:

> *The problem is almost never what you thought it was - that's why you're looking for it!*

Moreover, these results tell us something else: the version of the routine TAXPY in SCILIB is not unrolled, and that's why it's got the same crummy performance!

11   A Surprising Result

As is so often the case, we see only through a glass, darkly. I was rather disturbed at the results from HPM of an overall MFLOPS rating of 4. The standard LINPACK benchmark, which uses the same routines, gets a rating of

Table 7: Results using TAXPY, with extra unrolling

Routine	Seconds
TGEFA	0.180
TGESL	0.003

Table 8: CPU time for 1000x1000 (positive definite banded) matrix:

Tool	TGEFA	TGESL
(nothing)	5.429	0.002
(no vector)	124.158	0.373
HPM	5.450	0.002
FLOWTRACE	15.662	0.026
PERFTRACE	37.269	0.144
PROF	5.428	0.002
SCOUNT	6.205	0.002

Table 9: Timing for specialized LINPACK band factor/solve routines:

Routine	Seconds
SPBFA	0.002
SPBSL	0.001

150. Granted, we have some I/O and things, but is this reasonable? What could be different? The subroutine names? Nahhhhhh.... The data? Well, yes. Let's see, we've got a band matrix, and we're calling a routine that is appropriate for a full matrix. In other words, there are lots of computations that involve vectors that are mostly zero.

Let's take this a little further. First, let's change the data to use a full matrix. With no other change, if we run HPM, we get an overall MFLOP rating of 84. How can this be? The calculations are formally identical. Clearly, what's happening is that the Cray disdains to multiply by zero, and doesn't even count that as an operation! If we had more time, we could go in and time this code carefully, and see if we come closer to the 150 MFLOP rating. Instead, let's at least boost the size of the matrices up to 1,000.

Now let's try calling the standard LINPACK routines for this particular kind of (500 by 500) band matrix.

Well, that takes the cake, doesn't it? For this particular problem, we can get a stunning improvement in execution speed, if we'll simply use the most suitable algorithm for the given problem - in this case, switching from LINPACK's "general" routines SGEFA and SGESL to the special band routines SPBFA and SPBSL.

It might be time now for moral number 2:

Don't waste your time trying to improve the wrong algorithm!

12   Rules of Thumb

Benchmark the real program, not a 'toy' version. Small problems will not show how vectorization improves speed. If DO loops are shortened, or tolerances relaxed, or the number of iterations set to 1, then the true behavior of the

program will not show up, and one time operations like input/output or pre/post processing may dominate the report.

Nothing is 'linear'. A vector of length 64 doesn't take 64 times as long to process. However, it doesn't take exactly the same time as a scalar to process either. Timing a program that deals with vectors of length 500 doesn't necessarily tell you much about how vectors of length 5000 will be handled. If subroutine A is called once, for 1 second, and subroutine B is called 1000 times, for a reported total of 1 second, then the two subroutines probably do not really take the same total amount of time. The timing report is a number like T+fudge. So subroutine A maybe really takes 1-fudge seconds, but subroutine B takes 1-1000*fudge.

As the sample program should demonstrate, benchmarking programs may have difficulty monitoring a program with many subroutine calls, or a complicated structure. Each time an 'event' occurs, the benchmarker has to do something. This action itself takes time, and may result in inaccurate timings, particularly if the event being timed is brief. A well vectorized subroutine may be reported as being a time hog, simply because it is called very often.

Benchmarking should be done at one time, and then turned off. Users have been found running production codes with benchmarking on all the time. Some benchmarking inhibits vectorization, other options are very expensive in time, many generate reams of output.

Benchmarking is not precise. The timings of one run will differ from the next, varying with system load, compiler and loader choices, and so on. The important thing is to look for significant differences, 'relatively' large (10% say) and 'absolutely' large (more than 0.001 seconds, certainly). Differences of such magnitude are unlikely to be caused by the timing overhead or random fluctuations and can be attributed to differences in the programs you are comparing.

Consider worrying about memory bank conflicts. This is a difficult problem to attack in general, but easy to investigate. Sometimes, just making sure the first dimension of an array is odd will help a lot.

13   Contradictions

One of the most frustrating things about benchmarking, or using the Cray, is that you can't really rely on a simple rule. The more you know, the more you find out that a simple rule has exceptions. If you try to adjust the rule for those exceptions, then you will later run into interesting problems for which your rule is still inadequate. I'm not saying it's wrong to make rules; in fact, it's really important to try to summarize your experiences so you can begin to understand them. I'm just saying it's a complicated world, and you should get used to your rules being of limited use.

Consider, for example, the following series of statements. Each statement can be objected to, or criticized, and so a "better" statement follows. Of course there's no perfect statement that we can arrive at. It's better to try to remember that the simpler, earlier, statements are usually "true enough", but we have to

be prepared to think more carefully in a situation where they seem to be failing:

- *The Cray is fast!* - but of course, the real speed up comes not from the faster processor, but from the vectorization of DO loops.

- *The Cray speeds up DO loops.* - but of course, some DO loops have too low an iteration count, so the overhead of vectorizing actually slows you down. So,

- *The Cray speeds up DO loops that are heavily used.* - but of course, some loops don't vectorize, because so many lines of code are in one loop, one of them is bound to inhibit vectorization. So be sure to break up your loops into simple pieces.

- *The Cray speeds up DO loops that are simple enough for it to recognize as vectorizable and heavily used.* - but of course, some loops contain only one or two lines of code, and this can leave the processor half idle. So it's good to cram more calculations into one loop.

- *The Cray speeds up DO loops that are simple, but not too simple, and heavily used.* - but Of course, if a loop is too complicated, even if there are NO vectorization inhibiters, the loop may slow down because the processor doesn't have enough "on board" memory to keep all the temporary vector results.

- *The Cray speeds up DO loops that are not too simple, and not too complicated, and heavily used.* - but ...and so on and on!

SCILIB is a heavily vectorized library. Use it whenever possible. On the other hand, because we had source code some SCILIB routines, we were able to see that FORTRAN source can be faster than SCILIB. We even know why: unrolling of loops.

14   References and Help

At the PSC, brief documentation is available on the VMS front end through the HELP command. Documents are available in the VMS directory PSCDOC, and in the CFS directory /usr/local/doc. Examples of how to use various programs are in the VMS EXAMPLES directory, and in the CFS directory /usr/local/examples. Finally, the source code for programs is in the CFS directory /usr/local/bin or /usr/local/lib.

Taking LBENCH (the LINPACK benchmark program) as an example, you can type

```
HELP LBENCH
```

on VMS, to get a summary of information about the program. To access the document, you might type

```
        TYPE PSCDOC:LBENCH.DOC
```

on VMS, or on UNICOS, access the CFS copy by

```
        cfs get /usr/local/doc/lbench.doc
```

You could see the VMS directory of examples of how to use LBENCH by typing

```
        SETUP EXAMPLES
        EXAMPLES LBENCH
```

or on UNICOS you could get a listing within CFS

```
        cfs list /usr/local/examples/lbench
```

You could get the source code for LBENCH out of CFS by

```
        cfs get /usr/local/src/bin/lbench.f
```

In the PSC EXAMPLES directory BENCHMARK are examples of the use of PERFTRACE, FLOPTRACE and SCOUNT. Similar directories show the use of FLOWTRACE, HPM and PROF.

Runs of the benchmarking programs, and their source code, are available in the PSC EXAMPLES directories for EXPORTS, LINPACK, MFLOPS, NASKER, VECTORIZE, and WHSTONE.

In particular, Jack Dongarra's paper on the results of the LINPACK benchmark on various machines is available in PostScript form in the LBENCH example directory as LBENCH.PS.

The FORTRAN programs used to benchmark library software, as well as the output CPR files, are available in the various example directories for IMSL9, IMSL10, NAG11, and so on.

A brief writeup on FORGE is available in PSCDOC:FORGE.DOC. The FORGE online seminar is available during interactive use of FORGE. A copy of this information is available as a document in PSCDOC:SEMINAR.DOC. A book by John Levesque and John Williamson, of Pacific Sierra Research, that is based in part on their experience with FORGE, is *A Guidebook to FORTRAN on Supercomputers*, Academic Press, 1989.

Useful Cray Research, Inc, Manuals include:

- *CFT77 Reference Manual*, SR-0018C, Discusses FLOWTRACE, FTREF, PROF, HPM, multitasking, vectorization, and the various compiler directives An out of date version is available on the VAX as PSCDOC:CFT77.DOC

- *UNICOS Performance Utilities Reference Manual*, SR-2040A, Discusses FTREF, FLOWTRACE, PROF, HPM and PERFTRACE. An out of date version is available on the VAX as PSCDOC:PERFORM.DOC.

Table 10: What quantity does each benchmarking program report?:

Tool	Report
FLOWTRACE	CPU time spent in each routine.
FORGE	CPU time spent in each DO loop.
FTREF	The subroutine calling tree and COMMON block usage.
HPM	The overal MegaFLOP rating.
LOOPMARK	Which loops vectorize.
PERFTRACE	The MegaFLOP rating for each routine.
PROF	CPU time spent in small code segments.
RTC	The real time clock reading.
SCOUNT	Number of times each statement was executed.
SECOND	CPU seconds elapsed.
TIMEF	Wall clock milliseconds elapsed.

Table 11: Method of use:

Tool	Report
FLOWTRACE	Compile with option "-ef -a static", load, execute.
FORGE	FORGE preprocesses source code, analyzes log files after run.
FTREF	Compile with options "-exs -dB" to produce report.
HPM	"hpm executable" produces report after executable has run.
LOOPMARK	Compile with the option "-em" to get the listing.
PERFTRACE	Compile with "-ef -a static", load with "-l /lib/libperf.a", run.
PROF	Compile with option "-eD", load with "-g -l prof", run. then run prof.
RTC	Insert calls to RTC in source code.
SCOUNT	Preprocess source code with SCOUNT, then run for report.
SECOND	Insert calls to SECOND in source code.
TIMEF	Insert calls to TIMEF in source code.