Discussion and Correspondence

Software requirements of universities

by P. A. Samet*

The principal characteristic of the work load presented to a computer in a university environment is the preponderance by number of very short jobs, taking only a small fraction of the total time. The efficient and economic handling of this flood of small programs is a major problem and it is the author's contention that the software normally supplied by the computer manufacturers is unsuited to this type of load. The paper examines the needs of universities with reference to batch-oriented and multi-access systems.

1. Introduction

The writing of compilers and specialised operating systems is a major activity of many computing centres in universities. To the extent that this is done to gain experience such effort is very valuable: it is easier and better to teach techniques of which one has first-hand experience. Often, however, the motive for writing such basic software is simply the need to provide a good service to one's users and stems from the belief that the systems provided by the manufacturers are wholly inadequate for the needs of a university service. This belief, and the practice of having locally-written systems, is so wide-spread and persistent in universities that it appears relevant to enquire into the software needs of universities and why these are apparently not being met by the computer manufacturers.

2. The typical university workload

The main characteristics of computing in universities are the very large number of amateur programmers and the high proportion of small programs. One noticeable fact about these small programs is that they are each used only a few times, often indeed for only one 'production' run. There is also, in general, a small number of jobs that require considerable resources, either time or storage (or both); these programs normally are connected with the long-term research interests of the university and require the facilities of very large machines. Providing the physical resources are adequate, these large programs cause little difficulty to the management of a university computing centre. The big problem stems from the demand to handle the small programs.

I have just referred to the number of programmers. This is inevitably high. It is necessary to teach undergraduates and it is necessary for research students to have computing facilities. As there is a new intake of students each year this teaching is a continuing load. It is also important to realise that very few of these students are specialising in computing science, mostly their concern is to become mathematicians, physicists, engineers, economists, zoologists, linguists and the like. They are taught because it is essential for the successful prosecution of their studies that they have some acquaintance with computing methods. As more and more disciplines come to realise the uses to which they can put computers, so this need for teaching is expanding. In the United Kingdom we are at present not teaching anything like the number of students who will actually need these skills, although the number is going up rapidly. A few years ago almost all the students who were taught were post-graduates, whereas now it is becoming a requirement that undergraduates—often still in their first year—become sufficiently familiar with programming to use computers as normal tools in their regular course work. Almost all of the programming will be in a high-level language and this is, indeed, true of the vast majority of work done in any university installation.

The programs written by this large body of students tend to be very small. It should not be thought that they are only classroom exercises or the results of programming courses. More and more of these programs arise from their own work. This is as it should be, but it does mean that there will be many small jobs from this source. To get some idea of the size of the problem, it is instructive to see what the potential load would be if all the science based undergraduates in a university were taught to use computers in their first year and were able to use them as an adjunct to their normal work thereafter. In London University there are about 15,000 such students. If we assume that each student writes 20 programs per year on average, that each of these requires four runs before it is debugged, and that each run requires 15 seconds on a machine like the IBM 7090, we have a requirement for 5,000 hours of 7090 time per year, equivalent to two shifts every day of the year, and 1·2 million program runs per year. This figure means a daily load of 3,500 programs if the machine works a seven-day week. This is solely the undergraduate load, without counting the research students or the university staff. Extended to cover all students at colleges in Britain such a load would require the full-time equivalent of about four Atlas machines.

Many of the programs written in the course of research work by staff and postgraduates will also come in the 'small job' category. This is because, by and large, they do not handle vast quantities of input or output, nor do they normally require access to enormous files of information. Basically what they need is speed, and modern machines can supply this. The distribution of the work at the installation with which I am most familiar, the 360/65 at University College London, is shown in **Fig. 1.** I believe that this distribution is fairly typical of other installations. The information was collected over a period of eight months, with several hundred programs being handled daily. What is striking about these figures is that (just under) 50% of jobs run for less than 1 minute but took only about 8% of the total time, whereas only 1¼% of jobs took more than 20 minutes but accounted for 25% of the time. There is also a common belief that because scientific users do not have large input/output requirements university installations do not have any experience of handling bulk I/O. At UCL the daily card-reading load is more than 150,000 cards with more than 400,000 lines of output, and the work load on the machine is still going up steadily. The matching of input

* *Computer Centre, University College London, 19 Gordon St., London, W.C.1.*

and output on this scale is a major management problem, that I comment on later.

The figures of the previous paragraphs pin-point the university problem: how to handle a very large number of small jobs. This is the big task that justifies the big machine.

3. What the manufacturers supply

The usual justification for installing a very large machine is that one's organisation has some very large problems that have to be done at regular intervals. As I have tried to show above, the university justification is the single enormous problem of handling a large number of tiny jobs. The majority of the programs run at UCL do not *need* a 360/65; they could run equally well on a slower machine like a 360/40, only then we would need about three machines, dedicated to the small jobs, to cope with the volume, and still need a large computer to deal with the long-running jobs. It is certainly cheaper to have the faster machine. In short, we have to use a 'number cruncher' as a 'student cruncher'. In my experience the software that is currently supplied for large systems is woefully inadequate for the university type of work. The person who doubts this need only compile and run the FORTRAN program

```
STOP
END
```

(or its equivalent in any high level language) thirty times in succession. I have recently been told of one machine, newly installed in a British university, and according to its specifications roughly equivalent to a 7090, that would take an eight-hour shift to cope with this test! As large numbers of talented and experienced programmers work for the various computer manufacturers, it is worth asking why there is so much university discontent with their products.

First, I think, it has to be recognised that only a small number of computers, in absolute terms, go to universities. Indeed, there are not many large computers. The main effort, therefore, has to go into supporting the medium and small installations, most of which will work on a small number of repeated jobs. The software has to be designed to satisfy as wide a range of customers as possible. Mostly, the software supplied does this, but universities are not typical customers. For most users, what is important is the speed at which the compiled program runs, not the speed of the compilation, and in many instances high-level languages are not demanded by the users. I comment on this further in § 4.

Another reason, at present, is the concept of families of compatible machines. The attraction of compatible machines is that programs can be transferred from one configuration to another, with little or no trouble. The penalty, as I see it, is that compilers and operating systems are written to work within restricted minimum configurations. Gone are the days when you could have any size of store provided it was 32K, when you could choose the number of tape decks required provided you ordered exactly eight. Now, one may have a machine with a wide variety of sizes and peripherals attached, but the system only makes use of an irreducible minimum. It is rare that the facilities of the larger machine can be fully utilised, although there are certainly some systems that give the user a choice of trading store for speed. Further, it is the small machines of a family that are first available to the manufacturer's own software team, and it is not until the larger members are available that mistakes in strategy become apparent. This is especially so when peripherals are involved: what is of virtually infinite speed to

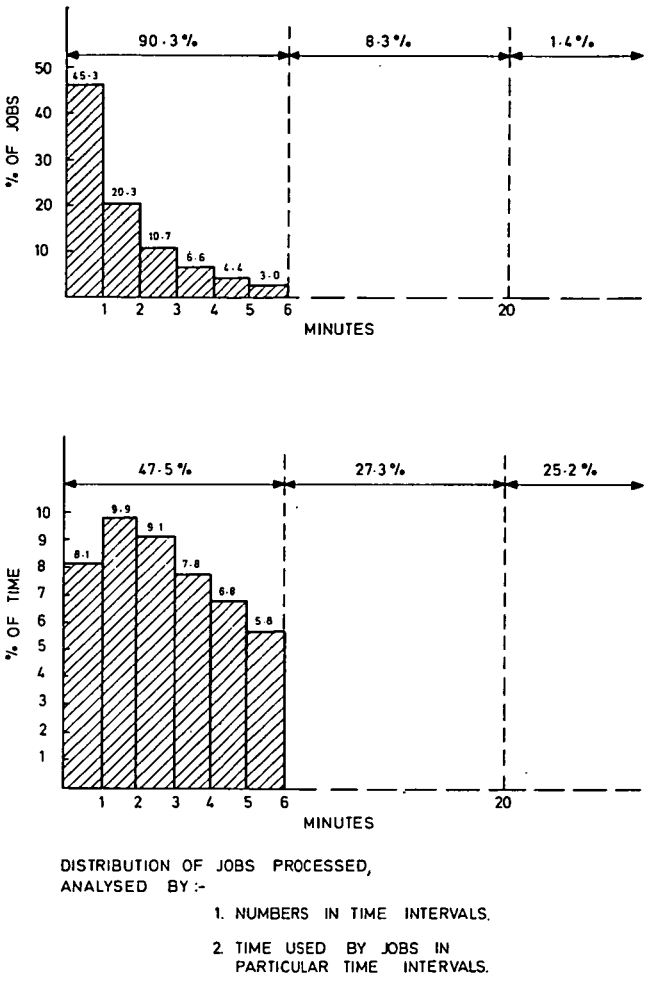

Fig. 1

the small machine often appears as of zero speed to the faster computer. Examples are the use of a small block size on tape, use of disks as if they are random access stores, failure to appreciate the effect of channel contention if several devices share the same control unit. Multi-pass compilers suffer terribly in this respect, yet they are a necessity on a small machine.

I am sure that another contributory factor is the very size of the software teams employed by the manufacturers. Communication difficulties abound, it is difficult to have adequate control over what is being written, and because a major piece of software is written by so many people it has to be broken into very many small segments with all the attendant inefficiencies of passing parameters between these segments. The software writers are, almost inevitably, remote from the needs of the customers and also from their own machines, which are normally operated on a 'closed shop' basis. As a result, the people who write the operating systems rarely see them working and are therefore unaware of many of these shortcomings. I can think of no other explanation for operating systems that refuse to recognise tapes and disks that are mounted and ask for them to be moved to other devices, for compiling systems that punch output onto a paper tape and then require the tape to be cut up for re-input in a different order, for systems that put 80-character blocks

(i.e. single card images) on magnetic tapes while employing packing densities of 800 bpi thereby giving an effective transfer rate which is about one-eighth of the nominal tape speed, for systems that require a minimum of 30 control cards for even the simplest job.

4. What do the universities need?

I have already pointed out that the main problems are:

(i) the large number of programmers, mostly interested in subjects other than computing;
(ii) the large number of tiny jobs.

Virtually none of these programs will be written in an assembly language, and this is, in fact, also true of the large programs. The principal needs would therefore appear to be:

(i) very fast compilers,

and

(ii) operating systems with minimum overheads.

I discuss both of these in the next sections.

4.1. Compilers

The arguments, indeed battles, of the early days of compilers, when many people had doubts about the technical feasibility of producing a working compiler, let alone one that generated good code, were a traumatic experience for the manufacturers. It became so important to produce good code that the compiling time was almost irrelevant. Techniques for writing compilers are now comparatively well understood (they even form topics in undergraduate courses) but the emphasis is still far too often on the production of good code. For the oft-repeated job this is, of course, necessary, but even here, I suspect, it is over-rated. What is essential is that frequently used subroutines are coded as well as possible; the efficiency of the main program hardly matters. It is vital that the correct method is used for solving the problem, a point that is frequently overlooked by users who only too often seem to believe that a high-speed computer makes numerical methods unnecessary. In a university environment, where most jobs live for only a few runs, the really important factor is how quickly the compilation can be achieved. Not that run-time efficiency can be totally ignored! Excellent FORTRAN compilers meeting this type of need have been produced by Purdue University (PUFFT) and University of Waterloo (WATFOR). These were written by very small groups closely in touch with their users and are considerably faster (by an order of magnitude) than other compilers for the same language available on their respective machines. True, the language implementations are somewhat restricted but not so as to inconvenience the large majority of users. Run-time efficiencies are quite acceptable. Another example, of rather earlier vintage, is Mercury Autocode. Very fast ALGOL compilers have been rare. One of the fastest was probably Whetstone ALGOL for the KDF9, whose run-time inefficiency was a computer manager's nightmare. Widespread acceptance of PL/I will surely depend, among other things, on the production of suitably fast compilers so that the language can be taught to large numbers of people.

It is relevant to remark that good diagnostics are most important in an environment where so few of the users can be classed as computer specialists. Such diagnostics must be present at compile time and preferably also at execution time. It is important that the run-time messages are made to refer to the source program, not to the compiled object program. This can be done; WATFOR certainly does it in great detail and this is one of the features that makes it so attractive to its users. All too often, though, the poor user is given a cryptic message which is sometimes followed by a core dump in octal or hexadecimal.

One feature of many modern systems is the ability they give the user to write parts of his program in different languages. There are occasions when this is a most valuable facility. Most university users, however, are monoglots, generally employing a restricted dialect of their chosen language. To impose the general system ('link editing', 'consolidation', 'composing' are some of its aliases) on all users causes a great waste of time. A system that can by-pass these overheads for single-language programs is highly desirable. To show what can be achieved in this direction, such a by-pass has been written for UCL by one of the IBM staff; whereas we previously had a minimum overhead of 25 seconds we now have a maximum overhead of 5 seconds. To appreciate the significance of this, remember that Fig. 1 shows that almost 50% of all jobs take less than 1 minute.

4.2. Operating systems

The principal requirement must be rapid and fully automatic job-to-job transition. For many of the programs that will be involved, any operator intervention, such as having to type initiation or termination messages, will take longer than the job itself. Even for the automatic systems it is essential that the overheads are kept down to the minimum. An inter-job time of, say, 5 seconds is negligible for jobs taking 30 minutes but is too much if an average job is processed in 30 seconds or less. There are very few manufacturers' systems that achieve such a low figure. Presumably this is because a general purpose operating system has always to be ready for anything. The WATFOR and PUFFT systems, mentioned earlier, achieve a very high throughput rate by organising their own job-to-job transition, taking milliseconds, rather than going back to the machine's operating system and incurring overheads of several seconds. They are able to do this because they are geared to a particular type of environment and only look for a restricted number of possible events. Again, using the UCL machine as an example, the test suggested earlier of 30 'zero-length' programs took 7·69 seconds as a WATFOR batch and this time included 6·47 seconds for loading the compiler initially. For comparison with the standard system, the total time for these 30 programs was 18 minutes 35 seconds using the system's link editor, and 10 minutes 43 seconds with the special fast link editor-loader.

The user's access to the machine is dominated by the control parameters that have to be supplied. Most of our users have standard requests from the system and so there is considerable advantage in providing a system that requires the minimum of information to be supplied by the user. In practice, this means very few control cards or characters. The minimum is two cards, one to identify the user and one to signify the end of his program, and so by implication the start of his data. Large packs of control cards may be in order for the professional programmer, although I would doubt it, but only act as a barrier for the average university user.

For most of the users in a university the immediate requirement is an execution run following straight on the compilation. Compilers therefore have to be of the 'load and go' variety, again without operator intervention between the separate stages. With the numbers of programs that will be involved we cannot afford, as a management problem, a system that requires program and data to be split, to be

matched again at a later stage. It is already enough of a headache to join input and output, without adding further complications. A consequence is that any unwanted data, e.g. that belonging to a program that has failed to compile, must be automatically ignored. This is, of course, essential anyway in any system that has automatic buffering of input and output. For the machine operators cards certainly have an operating advantage here over paper tape, in that several programs and their associated data can be placed in a card reader but not in a paper tape device.

Other requirements, which one would think were obvious were it not for the fact that systems are supplied without the facilities for meeting them, are the automatic identification of output, automatic marking of the end of a job on the printed output and starting a new page for the next program, and giving the user—and the installation management—a note of the time used for this run.

4.3. Batch systems and multi-access

Most important of all, however, is a system that allows very fast turn-round, say 2–3 hours at most and preferably much less, for most jobs. This is the area where multi-access systems have so much to offer, quite apart from any advantage that accrues from direct interaction with one's program (to my mind, an unnecessary facility in relation to most users' needs). However, even the despised batch systems can be made to run so as to give this very quick turn round. A very successful one, organised with several card-reader/printer data links, is the Case Institute System, working on a Univac 1107. Another, with rather more conventional hardware, is operated at the University of Waterloo where the turn-round for the short type of job that can be run with their own WATFOR compiler is *less than* 10 *minutes*. In most British universities, users with this type of program would be regarded as especially favoured if they get two turn-rounds per day. A decided gain from the hardware of the new generation of machines is the ability to have an input/output well on a disk or a drum (although first pioneered on Atlas with magnetic tape), instead of the physically recognisable batch of the 7090/1401 style that is generally used on tape-oriented machines. In particular, such a well can be organised to work with a priority system, so that the very short jobs can be moved forward in the queue, both for execution and output. It is impossible to do this economically on a sequential medium like magnetic tape. (Atlas gets away with it by multiprogramming, so that tape movement time can be used by another program.)

The prospect of multi-access systems is most important for universities. To be really useful we shall need machines with several hundred terminals *active* at any one time, not the penny numbers that are available at present. This appears to be a long way off, but need it be so? For most of our users what is required is the facility of a sophisticated calculating machine, without necessarily having access to a large and elaborate personal filing system, with password protection, etc. This restricted system should not be so difficult to implement, even if it is not as elegant and glamorous as the full-scale 'computer utility' concept. In fact, considering the current difficulties of several manufacturers over just this point, it might be a good idea to make the simplified system work first.

For the user, the multi-access system offers an entry to the machine without going through a central job-reception organisation, and gives hope for rapid turn-round. There is also an incidental advantage for the installation management in such a system, that there is no longer the need to tear up vast quantities of paper and then match them with the input documents. I would go further, and suggest that we look very carefully at what type of terminal equipment should be provided. Very many of the program runs, as we have seen, will be development. This means that on most occasions a program will differ only slightly from the previous version. The user's first action, more often than not, is simply to see whether there are any diagnostics, indicating that his program has failed. If so, he is only interested in the mistakes that he has made and has no further use for any remaining output. Why should he be burdened with it? It seems to me that a useful system would be one that allows short-term storage of files (program or data), provides editing facilities and an inspection system. Suitable hardware for the implementation of such software appears to be disks and CRT display consoles. Hard copy might be a problem, but could presumably be requested from the central system's printer. (I have also heard a suggestion that a Polaroid camera could be used to good effect.)

4.4. Program libraries

I have said that many of the programs that are written are individual to the student, pertaining to some particular problem in his work. Nevertheless, there are several standard processes that are required: for example, in the analysis of laboratory experiments a common need is for least squares curve fitting. The value of the computer as a teaching and research aid is considerably enhanced if there is a good library of easily accessible programs or algorithms. By this means the student can concentrate on what is important for his work, namely the interpretation of his results, rather than waste his time in removing stupid mistakes. He must learn to use complete programs just like other pieces of laboratory apparatus. It is debatable to what extent the selection and provision of such a library is the duty of the manufacturer. What must be the responsibility of the manufacturer is an organisation that allows such programs and subroutines to be called into use with the minimum of effort. If there is a system subroutine library, it must be possible to add to it in a reasonably simple manner, and similarly it must be possible to extend the library of complete programs that are available. I have known some systems where the library editing facilities were so complicated that their use required a major intellectual effort by highly skilled programmers to do even the simplest things.

Such system libraries will have to be implemented for batch-type and also for multi-access systems. My earlier remarks that few users would require access to large files refer to user-created files, not to system files.

5. Remarks on language and hardware compatibility

I have concentrated almost entirely on the type of work that is generated by students and research workers, but excluded the very large, continuing type of program. Although the emphasis has been on students, it is important to realise that much of the work done by university staff also has this 'short job' character, but is more durable.

One characteristic of university staff is their mobility. People move to other posts, they spend some time working in universities in other countries. Unlike people in all other types of employment, university staff take their work with them when they change their post. It is a source of con-

siderable annoyance to find that much of one's previous programming effort is of no further use, because the original programming language is not implemented at the new university or that the implementation, although allegedly of the same language, differs in important aspects. The same problem exists if a university has more than one computer available to its staff, or if it replaces a machine. Can we do no more than shrug our shoulders and say 'Well, that's life'? I would claim that agreed language standards are of the utmost importance to university users. By the same token, purely local languages—the provision of which seems to be a favourite university pastime—should not be used as the main working language of a university computing centre.

Just as important as the language compatibility is a certain amount of hardware compatibility. There is considerable interchange of programs and experimental data between workers at different universities. This makes it important for us to have ways of passing this information in machine readable form. Often this means compatible magnetic tapes but it also shows the need for compatible input/output equipment. The present jungle of mutually incompatible codes for cards and paper tape is to no-one's advantage and is totally unnecessary. It is also extremely costly, as new data preparation equipment may be needed when a university brings another machine into operation. For example, London University currently has Atlas, IBM 7094, IBM 360 and I.C.T. 1905 computers in operation, and will shortly add a CDC 6600. Four incompatible card codes will be in use, including three of the new 64-character sets. As all of these machines are available to users within the university there is considerable waste in having equipment that cannot punch cards for all the machines.

6. Concluding comments

I have tried to outline the characteristics of the computing load in a typical university and what software is needed to cope with them. In my opinion, manufacturers do not at present fill this need and so it is left to universities to write much of their own software. I suggest, however, that it is the manufacturers' own interest to look carefully at what the universities want. It is at college that young men and women get their first real introduction to computers. The impression they carry away with them from college can be most important in later years, when they are in a position to influence the installation of machines.

References

IRONS, E. T. (1965). A rapid turnaround multiprogramming system, *Comm. ACM*, Vol. 8, p. 152.

LYNCH, W. C. (1966). Description of a high capacity, fast turnaround university computing centre, *Comm. ACM*, Vol. 9, p. 117.

ROSEN, S., SPURGEON, R. A., DONNELLY, J. K. (1965). PUFFT—The Purdue University Fast FORTRAN Translator, *Comm. ACM*, Vol. 8, p. 661.

SHANTZ, P. W., GERMAN, R. A., MITCHELL, J. G., SHIRLEY, R. S. K., ZARNKE, C. R. (1967). WATFOR—The University of Waterloo FORTRAN IV Compiler, *Comm. ACM*, Vol. 10, p. 41. (This describes an earlier version of the compiler, not that for the System/360.)

Correspondence

To the Editor
The Computer Journal

Some computational notes on the shortest route problem

Sir,

There is a minor mistake in the procedures in the paper by Aarni Perko, published in the April 1965 issue of this *Journal*.

With regard to Bellman's optimisation principle, the iterative application of the procedures at each step gives an optimal solution for the nodes already considered. In general, in each iterative cycle the length of the route to a node i is calculated in an optimal way. If, however, the network contains isolated nodes, i.e. nodes which are not attainable from any other node, it can then be the case that after some iterations no nodes will be assigned to the array p. Considering this, the last two instructions with the data of the preceding iterative cycle will be executed wrongly.

In both procedures the lines

$d[jm] := x; p[jm] := im;$

should be replaced by

if $x < 99\,999\,999$ **then**
begin $d[jm] := x; p[jm] := im;$
end;

Provided that the arrays p and d are declared from $0 - n$ instead of $1 - n$, the additional instruction

$jm := 0;$

is sufficient.

As far as the computational refinement is concerned, it is useful as well to introduce an auxiliary variable for loc $[i + 1] - 1$ in the **for** clause.

Yours faithfully,

UWE PAPE

334 Wolfenbüttel,
Fontaneweg 4,
Germany.

3 February 1968

(*Further correspondence appears on pp. 172 and* 194)