SESSION X

LARGE ON-LINE SYSTEMS

RONALD E. ANDERSON, *University of Minnesota*, **President**

A large computer looks at the distributed computing problem

JOSEPH MARKOWITZ

The Cambridge Project, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139

The various ways in which a large computer can be of service to a small laboratory processor are discussed. The Cambridge Project and the Consistent System (a collection of programs, models, and data) are briefly described.

The on-line use of computers has grown enormously over the past decade, both in volume and sophistication. This is true for both large, time-shared computers and small laboratory machines. However, the growth in large computers and that in small computers has tended to follow separate paths of development. Further progress, however, will be accelerated for each if they interact—interact, that is, in the several senses of the word.

Researchers increasingly acquire “small” computers to assist them in the laboratory. Often the initial function for which such a machine is procured is “process control.” Typically, there is some repetitious task to be performed wherein a few indicators are to be monitored and a few actions need to be initiated, dependent on the state of those indicators. Automation of such tasks is clearly desirable: It nearly always yields fewer errors, and, in addition, it frees the researcher for more creative activities.

A decade ago such automation was routinely done by specially designed pieces of laboratory equipment. Specific logic building blocks—relay, vacuum tube, or transistor—were uniquely coupled with mechanisms to accomplish the specific job. Economies of scale have since prevailed to the extent that it is now more feasible to use a much larger, standardized, and programmable collection of logic—the small, general-purpose computer. A by-product has been increasing standardization of interfaces to peripheral mechanisms.

It has also been true that as the cost of such a small computer declined drastically, the performance improved drastically. It is easiest to illustrate this progression with Digital Equipment Corporation's PDP-8. Initially designated as the PDP-5, the machine sold for just over \$30,000. The newest PDP-8s sell for about 1/10 of that amount, are five times faster, and have a slightly more powerful instruction set. The newest darling of laboratory automation, the PDP-11, is even more powerful and attractively priced.

Thus, for a modest investment, a laboratory might

automate control of an operation and find itself with a general-purpose digital computer with capacity to spare. Sooner or later it occurs to someone in the installation that computers can do things other than control a single process: computers can control several different processes at the same time (real-time time-sharing); computers can store and retrieve experimental results (data management); and computers can make computations upon those results (data analysis). Burdened by cost-recovery procedures at the computer center, annoyed by the apparent lack of interest at the computer center in the problems of the small computer, and hampered by their inability to do direct data transfer at the convenience of the small computer, the custodians of the small computer embark on a “do-it-yourself” program. Generally, this program turns out to be of unanticipated complexity and unimagined cost. However, as the manpower and money can be acquired incrementally and the expertise acquired likewise, the project progresses with no end in sight.

To make “full use” of the computer, large and powerful—and relatively expensive—peripherals are needed. Sometimes these add-ons are incompatible with initial system specification and require field modifications to the mainframe. And sometimes the peripherals demand peripherals of their own.

As software develops for each small machine, there is duplication and little transfer from one installation to the next. “Reinventing the wheel” is the phrase most often applied to much software written at each local small machine installation. This should not be taken as an insult to the intelligence of the personnel. Rather it reflects their training and the inability of their colleagues, who have preceded them, to transcend the exigencies of a particular application and the idiosyncrasies of a particular machine. (In part these latter difficulties are unavoidable insofar as small machines cannot support the more desirable higher-level languages.)

CENTRAL vs REMOTE SOLUTIONS

Many of the facilities needed to complement the machine could be supplied remotely via telecommunications. As we see it, the needed facilities break down into three major categories: remote hardware peripherals (e.g., mass storage); applications software (e.g., complex data management and data analysis programs); and programming aids for the small machine (e.g., a higher-level language cross compiler).

Some tasks, such as large-scale data management and analysis, are probably best carried out by programs residing in the remote, large computer. Other tasks, such as rapid real-time experimental control, are best accomplished with programs resident in the local, small computer. For many tasks, however, it is not clear whether the controlling program should reside in the remote, large time-shared computer or the local, small computer. Resolution in cases of ambiguity will depend on the speed of connection and the cost of communication and computer services.

PROGRAMMING AIDS

Let us consider in more detail the programming aids which a large computer might provide for smaller ones. In this category we include those tools which facilitate the writing of programs—whether by programmers for users or by users themselves. These include language translators (compilers, interpreters, assemblers, and the like); editors; debugging tools; simulators; and a library of source code from which needed pieces can be borrowed.

HIGHER-LEVEL LANGUAGES

Provision of higher-level languages is extremely important. Not only does a higher-level language permit easier programming, but it provides a way of sharing experience amongst programmers who might have similar applications but quite different machines. Already, experience with small machines indicates the value of higher-level languages such as FORTRAN and BASIC. However, use of these languages is sometimes precluded for the smallest machines inasmuch as the compilers themselves cannot run in such a restricted environment—although the object code which they might produce could. The use of even more advanced higher-level languages (e.g., ALGOL, PL/1, LISP), which might be better suited to a particular application, would aggravate even further the problems of limited machine size.

We would stress the importance of standardizing such higher-level languages. As an example of the problem, certain things can be done in a particular implementation of FORTRAN that cannot be done in other implementations (and that are not part of the ANSI standard definition of the FORTRAN language). Programs containing these nonstandard features cannot

be as easily transferred among the community of users. Many of these applications, which can be programmed by trickery in FORTRAN, can be programmed standardly in a language such as PL/1. Thus the desire for standardization motivates our interest in some of the newer and more advanced languages.

SIMULATORS

Many minicomputer programs must be debugged on the computer in which they will eventually operate because their proper behavior is dependent upon precise timing characteristics of the particular computer and its peripherals. Most programs, or parts of programs, are less dependent on the details of specific hardware. These programs can be debugged on any computer if the computer can simulate the “target” machine. We need to develop a set of simulators for a variety of minicomputers that will allow the debugging of typical programs. It is feasible to develop simulators for a number of minicomputers because, currently, all minicomputers have one of a few basic types of architecture. A good simulator should be able to run a debugging program in addition to the program being debugged. Thus, the debugging facilities available to a computer user correcting his program on a “target” minicomputer should also be available to the user correcting his program at a remote installation. In addition, a simulator can provide the user with important information regarding the precise timing of his program. This is highly desirable when there is some doubt regarding the suitability of the real-time application.

ASSEMBLERS

Once a minicomputer simulator is available on a remote computer, it is possible to remotely assemble code for the minicomputer by simulating execution of the standard assembler available for the minicomputer from the manufacturer. Thus, we can simply cover the cases where available higher-level languages prove either inadequate for a particular job or unattractive to the user. While this solution is expedient, it is not optimal because the assemblers supplied for most minicomputers have limited capabilities. Many contain no macro facilities or extremely limited macro facilities. We need more powerful assemblers with good facilities for macro processing and more generalized string manipulation. As with simulators, it is possible to provide assemblers for several machines because of the underlying similarity in instruction sets among a number of minicomputers.

TEXT EDITING

Text editors in small machines are, for the most part, sadly lacking. They suffer from inadequate buffer size and unambitious design. A good contextual editor (one that can locate strings by content, as opposed to needing

a line-number specification) is indispensable. Most large machines have passable ones, and some large machines have excellent ones. Those with which we have had experience on Multics (EDM and QEDX) are powerful, while at the same time easy to learn. The TECO editor on the PDP-10 is another example.

Text editors are essential for source code modification, data entry, program preparation, and the like. The editors are doubly useful when they are teamed with a good formatted text output program—one that fills lines, adjusts margins, etc.

SOURCE CODE LIBRARY

Once language translators are available which can take higher-level source code and produce object code for a particular small machine, we can begin to think of maintaining a library of source code. Of course, this entails all the problems of any library: cataloging, retrieving, informing users as to what the library has to offer, etc. This library would contain source code both for “programs” and “subroutines” (a subtle distinction). Problems of certification, validation, and quality assurance would also have to be faced . . . perhaps on a scale heretofore unthought of.

Nonetheless, this library, as it accumulated, might well prove to be the most valuable, not only as a way of freeing the machine user from the tribulations of programming, but also as a way of sharing scientific techniques on a noteworthy scale.

One might ask why the library should contain the “source” code, rather than object modules for the variously relevant machines. There are three reasons, each important to understand.

First, the availability of source code insures that only the needed programs are compiled for a “new” machine—i.e., one for which a language translator has just been provided in the context of this resource—and that such programs can be available when they are needed for that new machine.

Second, it is most likely that the program as originally conceived by its author almost, but perhaps not quite, fills the bill. It is easier and, more to the point, more desirable for future users that the changes be made at the source level.

Finally, insofar as source code is so much more readable than object code, and given that documentation, no matter how thorough, is bound to be inadequate at some time, the source code is essential.

APPLICATIONS SOFTWARE

Yet another way in which a large computer can be of service to a small laboratory processor is by providing applications software—programs that run on the large computer and do jobs in the general area of data management and data analysis.

Certainly, a portion of each of these tasks could be done locally. However, several factors favor having a

larger, central machine do these tasks—provided that large central machine conveniently accepts data transferred from small ones. And by “convenience” we mean that the transfer is effected instantly (more or less) at the instigation of the small machine. Needless to say, the data transfer must be accomplished with a minimum of protocol and a maximum of accuracy.

DATA MANAGEMENT

Providing that the data transmission can be properly handled, then there is considerable merit in calling on the large central machine for data management and data analysis. For one thing, the newer standards of interactive data management, with a convenient user interface and the ability to handle complex relationships among data elements, are difficult and expensive to design and implement. The difficulty and the expense are both much greater if the data management package must be written in a relatively primitive language and run on a small machine. Once a good interactive data management system with a relational capability is available we can begin to fulfill the researcher’s dream of cumulative data management—being able to dynamically compare the results of this condition of this experiment with that condition of some other experiment. We dream of making these comparisons without looking up results in dog-eared lab notebooks; without hunting up rolls of punched-paper tape and deciphering the hieroglyphics on the leader/trailer; without similarly searching for a particular box of cards or a “Dectape” from the right time period. True, some of us have better filing systems than others, but even when we locate the data, reading it into the computer and reconstructing its format is a formidable task. There ought to be a better way—and there is.

Ideally, all the data should be accessible on-line with properly chosen and well-described formats. A powerful, interactive data management system should allow you to make comparisons even if the relevant variable is called “x” in one data set, “gender” in another, and “sex” in a third—whether coded 0, 1 or M, F. For other relevant variables, one should be able to transcend scale differences between pounds and grams, between meters and feet, and between Fahrenheit and Centigrade.

In addition to the coding complexities of such a data management system, the mass storage requirements alone argue in favor of a large, central machine for such activities.

DATA SHARING

When a large central machine is used by Es for their own data management needs, it becomes natural to think of that machine and its data management system as an ideal mechanism for sharing data between researchers, providing, of course, that there is provision for access control at a sufficiently atomic level. It seems clear that multiple, simultaneous access of shared data is

best provided by a large central machine. If the data base is distributed, each node must maintain a good deal more communication equipment than it otherwise might; each node must be more responsible, and receptive, to demand access by others (clearly at odds with the role of most laboratory computers); each node has to be more reliable; and, finally, if users of the distributed data base intend to compare across various of its pieces, either a lot of redundant transmissions or a lot of undesirable copying will take place.

DATA ANALYSIS

An equally powerful argument can be advanced for a certain amount of data analysis being done on the large central machine rather than on the small laboratory computer. This is more true, of course, if (convinced by the preceding section) one is already taking advantage of central data management. Even so, the complexity of certain computational algorithms and the demanded arithmetic precision argue for certain kinds of data analysis being done on a larger machine. . . . Then, once the data is there, why not take advantage of central data management and storage (the large central machine is perhaps well-described by an absorbing-barrier model of sorts).

Naturally, all the data analysis programs available on the large central machine must work appropriately with all the file types managed by the data management system. The data analysis programs must also work easily with the outputs of other analysis programs. The programs and data must all form part of a system that exhibits a certain amount of consistency.

THE CAMBRIDGE PROJECT AND THE CONSISTENT SYSTEM

Now, the Cambridge Project has spent a great deal of effort creating such a consistent system—called the Consistent System—on a large (and potentially central) computer. It is now operational on the Multics time-sharing system at M.I.T. and could provide a portion of the large machine services to smaller laboratory computers.

The Consistent System is a collection of programs, models, and data. All the programs of the collection

obey a number of ground rules. This ensures a loose coupling such that programs, models, and data tend to work nicely together whenever tenable; at the same time the rules are loose enough that new, previously unimagined programs could be added to the collection.

The collection now numbers in the hundreds of "programs." Some of these programs are large ones—for example, the Janus Data Management System or the Time Series Processor—and some are small—for example, a program which takes a vector mean or inverts a matrix.

All the programs work with and produce well-described data formats; in fact, one of the ground rules is that each program is obligated to check a file-type code before attempting to operate on a file of data. And it must code files of data which it produces.

The Consistent System probably could not have been assembled on a small computer. True, many of the individual programs could be made to run in such an environment. But, the system—which makes the whole greater than the sum of the parts—could not have been developed on a small machine. Nor, most likely, would the necessary critical mass of software have accumulated on any one small machine.

We feel that the Consistent System, on Multics, could fulfill a few of the support needs of small laboratory computers—namely, the applications software: the data management, data storage, and data analysis needs. The Consistent System could fulfill these needs if small laboratory computers were in communication with it. In general, they are not in communication now—that is an unfulfilled need. Of course, any laboratory computer that could emulate a terminal could achieve some communication . . . and, of course, Multics is on the ARPA network. Still, the communications need is largely unfulfilled. What other needs are unfulfilled? Nearly all of the others I have mentioned under the category of programming aids: higher-level language translators and compilers, simulators, assemblers, and debuggers.

A logical next step is to provide these missing ingredients and provide for adequate communications. This would be a project as ambitious as the Cambridge Project. Yet, it is one that needs to be undertaken if we are to realize the full potential inherent in that package of logic in the laboratory: the small, general-purpose computer.