DISTRIBUTED PROGRAMMING WITH $FLOW$

Seth G. Fearey
February 1980

Hewlett-Packard Co
Cupertino IC Operation

Technological developments in the information processing industry have done wonders for reducing costs and increasing throughput. But to most of the users, the computer remains a black box to be controlled only by trained professionals using unintelligible languages."

The spread of Distributed Processing exemplified by the HP300 has brought the black box closer to the user and often allows him or her to sign on, load data and run selected programs . But the user is still constrained to the systems installed and maintained by professional programmers, and programmers still operate in batch mode - submit a request and wait three months for a response.

Distributed Programming is the next logical step toward greater utilization of the computer and higher productivity.

****

I. Concept of Distributed Programming

Definition. In essense, applying Distributed Programming means making the power of the computer available to the users.

In a Distributed Programming environment programs will be written, run and maintained by users in all professions at all levels, from accounting clerks to production managers.

A. Benefits/Costs

Many of the benefits of Distributed Programming are obvious:

1. Clerical productivity and accuracy improve.

2. Professional programmers are relieved of small, nuisance projects.

3. Reports are tailored to the user's desires instead of being tailored to fit the time the progammer has available.

There are also secondary benefits:

1. Red tape in the form of Program Requests, Maintenance Requests and programmer scheduling is eliminated.

2. Users feel more in control of the computer.

3. Users develop a greater understanding of the complexities of programming and of what the computer can and cannot do.

4. Users have a new outlet for creativity, improving job interest.

At the same time, there will be drawbacks:

1. Time must be invested in user training.

2. The demand for user labor may increase as users demand more information and take on tasks normally performed by keypunch operators, graphics artists and secretaries.

3. The need for terminals will increase dramatically.

B. Prerequisites for Success

Distributed Programming will not get started without planning and support. In fact, those who have the most to gain may fight the hardest against letting the unsophisticated user on the computer. Data Processing professionals will fret over "SECURITY" and managers will say: "That's not part of your job - we have secretaries and programmers to do that."

Assuming management is willing to give Distributed Programming a try, there are other prerequisites:

1. Software must be is "user oriented", ie. friendly, on-line and flexible. The example of $FLOW$ is described in detail below.

2. User training. Users will have to be taught how a terminal works, how to log on, and how to operate the user software. A good training program will assuage the fears of the professional programmers regarding security.

3. Interested users. Distributed Programming cannot be forced on the uninterested. Those with the curiosity will catch on quickly and inspire others to give it a try. All users will need a willingness to experiment and go on despite failures.

4. Distributed Processing. If each functional area has its own computer the other areas will not be as afraid of meddling by non-professionals.

5. Accounting. The accounting system should not penalize users for their connect time. Distributed Programming will flourish most easily in an environment where computer time is considered a "free resource."

II. The Example of $FLOW$

A. $FLOW$

A good example of a language appropriate for Distributed Programming is $FLOW$, a package available from the Palo Alto Group, which I have been using for the last year.

$FLOW$ is a self-contained, financial report writing language with tremendous flexibility. Applications at Hewlett-Packard range from marketing forecasts to production models to financial planning. (Two examples are described later.)

B. User Oriented Features

Using $FLOW$ as an example we can see the features which separate user oriented languages from traditional languages.

1. Friendly. The language must be on-line and interactive. $FLOW$ guides the user step by step through multiple option prompts, ex:

MODIFY DEFINITIONS(1),REFORMAT(2),DISPLAY PGM(3),INPUT NEW PGM(4), INPUT VALUES(5),COMPUTE REPT(6),SAVE PGM(7),LOAD PGM(8), OR DONE (CR)?

The selection of an option by entering a single number or letter, takes the user to the next level prompt and on down until the desired results are achieved.

2. Fault Tolerant. A user oriented language should prevent programming errors from occuring before the program is ever run.

There are two types of errors: the illogical and the improbable. In $FLOW$ illogical errors generate an error message and return the user to the previous prompt. For example:

UNBALANCED PARENTHESES. THERE ARE 2 MORE CLOSING THAN OPENING ()'S.

Improbable errors, such as changing a calculation in a report and not saving the new version result in warnings which can be over ridden only by a forceful response, ex.:

W601* WAIT! YOU HAVEN'T SAVED THIS VERSION OF YOUR REPORT YET! DO YOU WISH TO SAVE IT ON A SAVE FILE BEFORE SCRATCHING IT FROM THE WORKING FILES (Y,N)?

In addition, it is almost impossible to crash the program, an unnerving experience for new users of the computer.

3. Default Options. A user should be able to quickly generate output of acceptable quality. $FLOW$ has a great number of report formatting options, all with default values. The user is only required to set up the equations and enter data; $FLOW$ centers titles, arranges page breaks and numbering, underscores columns when a total occurs and more.

4. Self Documenting. A user oriented language should contain built-in documentation. Unraveling an unfamiliar program written in a conventional "user" language like BASIC, is difficult and time-consuming. In addition, users are generally not motivated toward documenting their work.

$FLOW$ listings are relatively straightforward. Variable names can be more than 50 characters; listing show both variable names and line number references (see examples). Listings also show when the program was last computed and modified.

5. Minimal Training. Users are not interested in spending several weeks learning a language. After a couple hours of formal training, users should then be able to use the language to teach themselves what they need through trial and error or a quick scan of the manual.

The many options in the $FLOW$ prompts encourage users to try them all out. When mistakes are made the user can return and reset the options or recall the last version saved.

While the above is not a comprehensive list of desireable features, the above items should be fundamental to all user oriented langues.

III. Examples

A. Using the computer to perform routine calculations.

The first example is a $FLOW$ program I wrote to allocate expenses to different locations. The versions attached are condensed to fit on a single page. The real program includes some 40 departments and 6 allocations. To do the same calculations by hand a clerk would work out the percentages on a calculator then multiply them by the amount to be allocated - a one hour task if it foots the first time.

The initial version of this report took only 15 minutes to set up. The final version is the result of another half hour's tinkering with the format. How long would it have taken in COBOL or BASIC?

A clerk now has total resonsiblity for this program, running it once a month and updating it as locations change or other expense items added.

A copy of the program listing is also attached to give you a better idea of what I mean by self-documentation.

B. Using the computer for analysis.

The second example was written by a clerk who had never worked with a computer before being introduced to $FLOW$ a year ago. Before writing the program she prepared the report by hand on a spread sheet. The accuracy of the calculations had to be checked several times, leading to erasures and a rather messy final version.

While the calculations are not terribly complex the report will always foot so the user can concentrate on the quality of the inputs instead of running endless tapes in hopes of hitting the same total twice in a row.

IV. CONCLUSION

I have attempted to make two points:

First, there are advantages to allowing users to become programmers, what I have called Distributed Programming. Benefits range from increased productivity to improved job satisfaction.

Second, in order to develop Distributed Programming we need new languages which are user oriented. The development of such languages will be difficult because users are not willing to devote their lives to understanding the computer.

Nonetheless, the trends are clear: home computers, robotics and talking television controls. As our machines become more flexible we must also become more flexible.