LA-UR -82-2059

CONF-821002--2

LA-UR--82-2059

DE82 019559

Los Alamos National Laboratory is operated by the University of California for the United States Department of Energy under contract W-7405-ENG-36.

TITLE: EXPERIENCES WITH DISTRIBUTING GRAPHIC SOFTWARE BETWEEN PROCESSORS

AUTHOR(S): Griffith Hamlin
James E. George

SUBMITTED TO: IEEE Computer Society, The Third International Conference on Distributed Computing Systems, Miami/Ft. Lauderdale, FL October 18-22, 1982

DISCLAIMER

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes.

The Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy

Los Alamos National Laboratory
Los Alamos, New Mexico 87545

FORM NO. 836 R4
ST. NO. 2629 5/81

EXPERIENCES WITH DISTRIBUTING GRAPHIC SOFTWARE BETWEEN PROCESSORS

Griff Hamlin and James E. George

Los Alamos National Laboratory
Los Alamos, NM 87545

ABSTRACT

Software to aid the distribution and coordination of tasks between different processors was developed to distribute applications written in Fortran. This development led to the discovery of problems unique to Fortran and to interesting practical solutions. Two graphical applications were distributed to a variety of machines and machine pairs: CDC 7600-LSI/11, Cray-LSI/11, VAX 11/780, and Apollo. These distributions pointed out several parameters such as the use of Fortran COMMON, communication parameters, and processing capabilities that can affect the successful distribution of applications.

Introduction

Los Alamos National Laboratory's Central Computing Facility (CCF) comprises several computers connected in a network: 4 Cray-1s, 4 CDC 7600s, 2 CDC-Cyber 73s, one CDC 6600, and VAX-11/780s connected as distributed processors at remote sites. Access to the CCF is accomplished via terminals connected to the network or to the distributed processors; the network supports 1200 low-speed terminals (300 baud), 32 medium-speed terminals (1200 baud), and 400 high-speed terminals (9600 baud receive and 300 baud transmit). There are also several hundred intelligent work stations composed of a terminal and a LSI/11 processor.

The rapid explosion of the microprocessor technology in the next few years will result in the incorporation of local processing into many computer terminals. Today's present "dumb terminal" will either be upgraded to an intelligent work station by the addition of local processing or be replaced with a commercially designed intelligent work station.

Our goal was to provide a software development environment that allows Fortran application programs to execute partly on a large host computer and partly on one of these programmable intelligent terminals. Ideally, the various software services commonly provided the programmer of a single processor should also be provided the programmer trying to program an application on two or more distributed computers. One approach to providing these services is to extend the programming language's procedure call to include calls to modules on remote computers. This solution is appealing because it hides the multiple computers and no new mechanisms need be learned by the programmer. This approach, called the Remote Procedure Call approach, has been tried at least three times[1,2,3]; these implementations never became widely used outside of their own environment.

In our current implementation of this approach, we attempted to discover what problems hindered their use in a production environment and what changes would be necessary to obtain productive use.

The applications at Los Alamos tend to be large programs written in Fortran. These applications have been implemented over several years, and it is not uncommon for the source files to be 20,000 to 100,000 lines. This multi-man-year investment precludes the re-implementation of these applications in other languages. Thus, our software was implemented for the Fortran programmer.

Other Systems

One implementation of the Remote Procedure Call approach is the ICOPS system developed at Brown University.[1] The program modules are declared by the programmer to be either host-resident, satellite-resident, or movable. A host or satellite language processor maintains

symbol table information about each module. No references to global variables across the computer boundary are allowed. Prior to the normal linking phase, the object modules are scanned by a link edit preprocessor written for the ICOPS system. This pre-linker changes references to remote subroutines into references to the ICOPS run-time system. The link editor includes the symbol table information from the object modules in its output. This information can be used after execution has begun, thus allowing the user to interrupt execution, specify a change in the configuration, and resume execution. Only modules originally declared to the system as movable may be moved between computers after execution has begun. This ability is useful for reacting to changes in the host's workload. The actual mechanism of movable modules utilizes a copy of the module in each computer. The ICOPS run-time system activates the dormant copy and deactivates the other copy.

Another implementation of the Remote Procedure Call approach is the CAGES system developed at the University of North Carolina.[2] The program modules, written in PL/1, are declared by the programmer to be either host- or satellite-resident. There are no movable modules as in ICOPS. In addition to remote procedure calls, CAGES supports remote global data references and remote exception handling. Prior to compilation, the source modules are scanned by a preprocessor written for the CAGES system. This preprocessor changes remote procedure calls into calls to the CAGES run-time system and generates symbol table information describing the procedures and global data items that are not structures. For structured data, stub procedures are generated by the CAGES preprocessor; these are compiled and linked with the user-written modules. At run time, the symbol tables and stub procedures are used by the CAGES run-time system to handle all inter-computer calls and data references.

Recent work in configurable programs includes the Emissary system at Carnegie-Mellon University.[3] Although not yet implemented, the Emissary system design allows remote procedure calls among several distributed computers and remote exception handling, but no global data. The design includes considerations for preserving the same semantics as local procedure calls, even in the face of crashes of one or more of the computers involved. It also includes preservation of strong type-checking of the parameters of remote procedure calls; neither of the previous systems addressed these issues. The source language used is Mesa [4] a strongly typed descendant of Pascal. The Emissary design used as its target implementation system a homogeneous collection of computers connected by an Ethernet communications link.[5] Thus, it sidestepped the problem of data translation found in heterogeneous computer systems.

Our Distributing Software

Our implementation of the Remote Procedure Call approach is Fortran based, with a preprocessor as in the CAGES system. It supports inter-computer SUBROUTINE calls and inter-computer references to named COMMON data. To distribute a program between two computers, three steps are necessary. First, given a Fortran application program, the programmer must decide which of the subroutines in the application should execute on the satellite processor and which on the host. This will mean that some subroutines contain CALLs to other subroutines that are not on the same computer. All such remotely called subroutines must be specified to the distributing software. This specification consists of the subroutine name and a description of its parameters and COMMON areas, as described below.

Second, the preprocessor is run. The result of this will be the production of two files, called HSTSRC and SATSRC. These are Fortran source files that must be compiled on the host and satellite, respectively, and linked into the application.

Third, the programmer links together the application main program and host-resident subroutines, HSTSRC, and the run-time library that is provided. The result of this linking will produce the executable part of the application that resides on the host computer.

On the satellite computer the process is similar. The satellite run-time library, the application subroutines, and SATSRC are linked together, producing an executable module on the satellite.

Every subroutine that can be CALLed across the two computers must be specified to the distributing software. This specification consists of the nonexecutable Fortran statements extracted from each such subroutine. These statements describe the data type and array sizes for all parameters and COMMON areas in the subroutine. These declarative statements have been expanded to allow the programmer to specify variables that contain character data, and arrays whose size is an expression

containing other parameters and COMMON variables. That is, the array size may change at run-time, up to a maximum declared size. In addition, there are specification commands to specify which computer each subroutine runs on, which of each subroutine's parameters are only used (value never set) and which are only set (values never used), and whether or not the subroutine should execute in parallel with its caller.

Experiment 1 -- Distributing Draw

Two existing graphics applications were distributed onto various pairs of computers using the system. The first, DRAW, is an application program for producing high quality two dimensional charts and graphs. It provides an intelligent interface so that a user is not required to write programs but answers questions and enters the appropriate data. DRAW is implemented on the CDC computers at Los Alamos. The CDC-7600 version was chosen to distribute between a DEC/LSI-11 and a CDC 7600.

DRAW consists of two parts, the interaction handler for commands and editing and the display generator; the display generator is implemented using DISSPLA.[6] The natural split was to move the interaction handler to the LSI-11. The particular distribution of the interaction handler was accomplished in five different ways and measurements were taken for each method.

Version 0

This is the original version of DRAW in which all of the software runs on the 7600.

Version 1

The interaction handler was moved to the LSI-11 and the communication was defined using the distributing software. In this particular case, if COMMON blocks existed, all of the data in COMMON blocks was transmitted each way. During this process, we discovered many problems of the use of COMMON and its effect on performance on distributed systems. This discovery led us to study the use of COMMON and to carefully control how much is transmitted each way. Figure 1 shows the time needed to modify a typical graph.

The measurements in this and all other figures are averages of several program runs at different times and days on our time-shared VAX and Cray computers without any special priority treatment.

EXPERIMENT 1: ELAPSED TIME TO MODIFY A GRAPH (AVERAGED OVER SEVERAL RUNS)

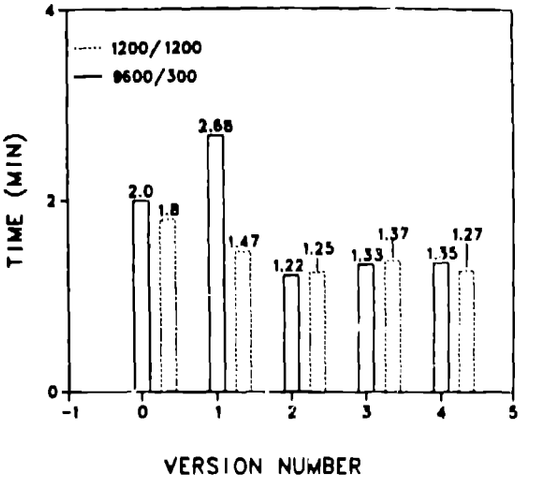

Figure 1

Figure 1 also shows two sets of line speeds; the in-house lines are 300 baud from the terminal to the host and 9600 baud from the host to the terminal. Additional dial-in lines are 1200 baud each way.

Version 2

The interaction handler was distributed to the LSI-11 as in Version 1. In this version, the host transmitted all referenced COMMONs to the LSI-11 but the LSI-11 only transmitted changes to the COMMONs back to the host. Performance is illustrated in Figure 1.

Version 3

The interaction handler was distributed, the host transmitted only the variables in COMMONs that contained data, and the LSI returned all changed areas of COMMON. Performance is illustrated in Figure 1.

Version 4

The interaction handler was distributed, the host only transmitted the variables in COMMON that contained data, and the LSI only returned the areas of COMMON that were changed and contained data.

Version 5

Version 4 was distributed to a few users and they requested that graphic tablet input be supported on the LSI-11 as an alternate input for digitizing data curves. Figure 2 illustrates the performance with a 9600/300 line and with a 9600/9600 line.

EXPERIMENT 1: EFFECT OF TIME SHARING LOAD ON TIME TO DIGITIZE A CURVE

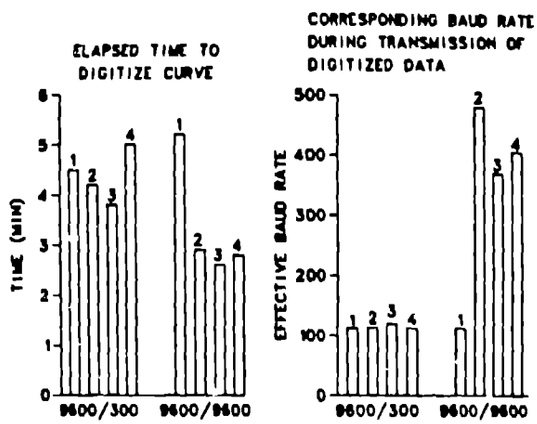

Figure 2

Experiment 2 -- Distributing MOVIE.BYU

MOVIE.BYU[7-9] is a display program for three-dimensional polygon and polyhedral models. It can produce line drawing and continuous tone monochrome or color images. It is written specifically to be a postprocessor for finite element analysis. MOVIE.BYU runs on several computers and the implementations for our experiments are the Cray and VAX 11/780 versions with distribution to the LSI-11.

Version 0

This is the production version of MOVIE.BYU that runs entirely on the Cray; Figure 3 shows the elapsed execution time and Figure 4 shows the execution time for the hidden surface calculation.

MOVIE.BYU ELAPSED EXECUTION TIMES
(INPUT DATA: 6500 ELEMENTS, 1700 NODES)

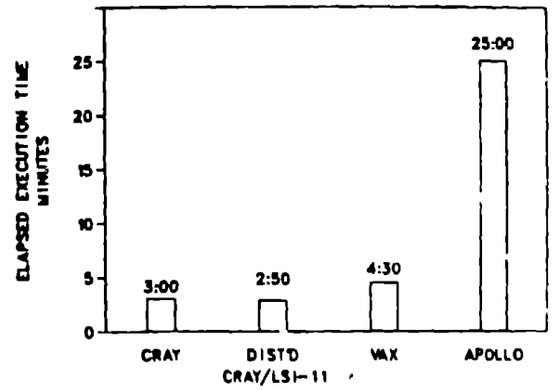

Figure 3

MOVIE.BYU SURFACE CALCULATION
(FLOATING POINT)

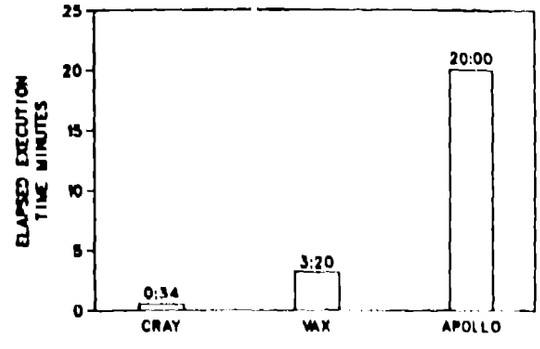

Figure 4

Version 1

All of the interaction handler is distributed to the LSI-11 with all of the computation left on the Cray. Performance is shown in Figures 3 and 4.

Version 2

All of MOVIE.BYU is implemented on the VAX 11/780. Performance is illustrated in Figures 3 and 4.

Version 3

All of MOVIE.BYU is implemented on the Apollo; performance is illustrated in Figures 3 and 4.

Observations and Conclusions

General Observations

We observed, while using this system, that it was somewhat tedious to describe to the preprocessor the needed information about all of the remotely called subroutines. Clearly, the preprocessor can automatically scan the source code for this information. Until the distribution can be accomplished less tediously, we believe that none of these distribution systems will be widely used by application programmers in production environments.

Debugging a distributed program requires some understanding of the underlying distributing software in order to interpret inter-computer communication to determine which computer caused the failure. Maintenance of distributed software seems to vary as some power of the number of computers involved rather than the sum. Also, fixing an error may mean a preprocessor run, two compiler runs (one on each computer), two linker runs, and finally execution of the application again. This process is time consuming and slows down the implementation process. Each step is done manually on our system; automation of this process is needed before a wide use in our environment will occur.

Inter-Computer Communications

As in previous studies,[10] we observed that our very slow (300 baud) data rate from the satellite to the host computer limited what applications could be distributed and still show an improved user productivity. Much of the time saved by the user in having instant interactions with a satellite computer was lost when the user had to wait for information to be sent to the host computer. The net result of this loss was often only a small reduction in total elapsed execution time. The effect is aggrevated by the "burstiness" of our inter-computer communications.

In typical time-sharing systems, transmission takes place one character at a time as the user types on his terminal and is thus overlapped with his typing. Our experimental applications tended to run for long periods of time on the satellite computer, interacting with the user locally. Then, after these interactions, rather large amounts of data needed to be sent to the host for processing. This resulting transmission time was annoying; Figure 1 illustrates this effect. In this Figure, Version 1 (9600/300) took longer than any other version; in this case, the satellite transmitted all of the COMMON data back to the host. We conclude that our "bursty" traffic favors packet switching communication techniques and time-sharing of high bandwidth communications channels such as are found in many local networks. Also, we observed more communications from satellite to host and less from host to satellite when running a distributed program than when running the program entirely on the host and using the satellite only as a terminal. Our time-sharing host computer system is set up to provide most of the terminal bandwidth in the host-to-satellite direction (9600 baud) and only a low bandwidth (300 baud) in the other direction. This does not appear to be the right choice for distributed programs. Rather, equal division of the bandwidth is needed. In fact, execution of our distributed applications over a 1200/1200 baud dial-up connection took only 10% more elapsed time than distributed execution on a 9600/300 baud line.

Figure 2 illustrates this transmission problem using Version 5 of the DRAW experiment. In this case, a curve with 16 data points is digitized locally, the data is transmitted to the host, and the resultant graph is drawn. The elapsed time curve shows that the best and the worst performance occurs with the higher line speeds; other factors are dominating the elapsed time.

The effective transmission rate of the digitized data is also shown in Figure 2. This effective rate is dependent on at least three factors: the communication line transmission rate, the size of the transmission packets or buffers, and the swapping time on the host time-shared computer.

On our test host, the packet sizes are 80 bytes for input to the host, and each packet of 80 bytes results in one swap. The best performance at 9600/9600 was 24 seconds for the data transmission, i.e. 14 packets and 14 swaps. Thus, an approximate swap time is 24/14=1.7 seconds. The total transmission time at 9600 for 1100 bytes is 1.2 seconds. With a longer packet (1100 bytes), the best transmission time expected would be 1.7+1.2=2.9 seconds vs. the best observed of 24 seconds. In the worst case, the transmission time was 95 seconds, or 95/14=6.8 seconds swap time. The expected worst performance with a longer buffer would be 6.8+1.2=8 seconds at 9600/9600; this is 3 times faster then the best performance observed.

For 300 baud input, 1100 bytes requires 36 seconds and the swap time is 1.7 to 6.8 seconds, best to worst. These data result in an expected performance range of 37.7 to 44.8 seconds. This range is 2 to 3 times faster than the observed data for 300 input and slightly faster than the average performance at 9600.

The curves of Figure 2 suggest that packet or buffer size is an important attribute in effective bandwidth of our network. Implementors should carefully evaluate the effective bandwidth to intelligent terminals by manipulating the various parameters such as line speed, packet size, and swap time.

Inter-Subroutine Communication

Perhaps the most serious problem with production usage of this type of distributed programs is our observation that many of the existing Fortran applications at the Laboratory indiscriminantly use COMMON areas to communicate among subroutines. MOVIE.BYU, for example, passes all of its data among its 68 subroutines using 49 different COMMON areas and no parameters. This communication mechanism is understandable as there is no penalty for communicating by shared memory in a single computer. Distributing some of these subroutines to a satellite computer causes much more communication to take place than is really required because of the lack of information-hiding in the structure of the subroutines and their patterns of accessng COMMON areas.

The most common example of this problem was using a few elements out of a large array. The next most common example was using only a few variables out of a large COMMON area. Curing these communication problems requires an understanding of the structure of the application code and modifying it, which is extremely time consuming. Transmitting the needed parameters and COMMON variables on demand between the host and satellite as it is referenced is a possible solution; implementing this would require compiler and operating system changes in most situations. Also, the user of a uniprogrammed satellite computer would see the demand paging communications time.

All of these problems suggest that distributed applications must be designed with distributing in mind, even though the distributing software allows the application source code to be independent of the distribution. The aim should be to make the application user unaware of whether he is running on a single computer or distributed computers. Structured programming techniques and information-hiding concepts using accessing functions should help in reducing inter-module communications. However, the Fortran language does not encourage this type of programming.

The editor scanning time, seen in Figure 5, is 12.5 times that of the VAX on the first scan and 7.5 times on repeated times. However, Figure 6 illustrates that the compile speed of the Apollo is 3.3 times that of the VAX. Compiling and editor functions are quite similar and one would expect these ratios to be similar; apparently, more effort has been spent on the compiler than the editor.

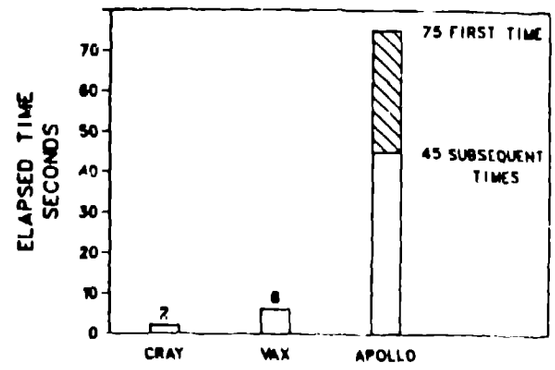

Figure 5

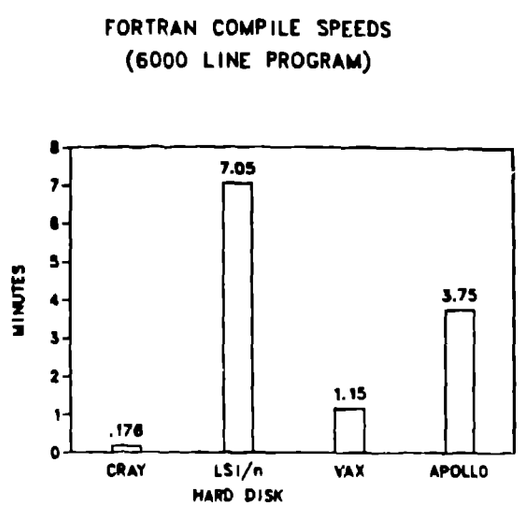

Figure 6

Figure 4 compares the execution on the hidden surface calculation that is predominantly floating point calculations; the ratio is 6.25 times longer for the Apollo. Figure 3 illustrates that the Apollo is 5.5 times slower on the entire MOVIE.BYU application run.

This suggests that one should not distribute inappropriate items to the intelligent work stations. It does not make sense to try to compete with the floating point processing of a Cray-1; it does make sense to compete with the interaction of the time-shared Cray-1. Improved interaction requires high speed communications between the host and the satellites for the data packets desired by the applications. It appears that desk top processing power is no longer the weak link; communications is the weak link.

References

1. A. van Dam, G. M. Stabler, and R. J. Harrington, "Intelligent Satellites for Interactive Graphics," Proceedings of the IEEE, Vol. 62, 4, 1974.

2. Griffith Hamlin, and J. D. Foley, "Configurable Applications for Graphics Employing Satellites," Computer Graphics, 10, 2 (1976).

3. B. J. Nelson, "Remote Procedure Call," Department of Computer Science, Carnegie-Mellon University, 1981.

4. J. G. Mitchell, W. Maybury, and R. Sweet, "Mesa Language Manual," Xerox Palo Alto Research Center, report SCL-79-3, 1979.

5. R. M. Metcalfe, and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," CACM, 19, 7 (1976).

6. DISSPLA User's Manual, Integrated Software Systems Corp., San Diego, California, 1981.

7. B. E. Brown, "GRAPE - A General Purpose Display Program for Three-Dimensional Finite Element Models," University of California, Lawrence Livermore Laboratory.

8. B. E. Brown, "Computer Graphics for Large Scale Two- and Three-Dimensional Analysis of Complex Geometries," Computer Graphics, 13, 2 (1979).

9. H. N. Christiansen, and M. B. Stephenson, "MOVIE.BYU - A General Purpose Computer Graphics Display System," Proceedings of the Symposium on Applications of Computer Methods in Engineering, University of Southern California, Los Angeles, Vol. 2, 1977.

10. J. D. Foley, et. al., "Graphics System Modelling: Verification and Applications," Department of Computer Science, University of North Carolina, 1975.