01 Jan 1990

HIGHLAND: A Graph-Based Parallel Processing Environment for Heterogeneous Local Area Networks

Ralph W. Wilkerson
Missouri University of Science and Technology, ralphw@mst.edu

Douglas E. Meyer

Recommended Citation

R. W. Wilkerson and D. E. Meyer, "HIGHLAND: A Graph-Based Parallel Processing Environment for Heterogeneous Local Area Networks," *Proceedings of the Fifth Distributed Memory Computing Conference, 1990*, Institute of Electrical and Electronics Engineers (IEEE), Jan 1990.
The definitive version is available at https://doi.org/10.1109/DMCC.1990.556277

HIGHLAND: A Graph-based Parallel Processing Environment for Heterogeneous Local Area Networks

Douglas E. Meyer
Ralph W. Wilkerson

Department of Computer Science
University of Missouri - Rolla
Rolla, MO 65401

Abstract

HIGHLAND, a distributed-memory parallel processing environment for heterogeneous local area networks, has been designed and implemented. Designed as both a teaching and a research tool, its purpose is to provide an effective mechanism by which a number of networked UNIX* workstations, dissimilar in both vendor and performance, can be directly manipulated as a single, unified, multiprocessing system. Utilizing the MIT X-Windows environment, HIGHLAND supports a highly interactive graphical interface through which a programmer can create, modify, and control complex systems of communicating processes. The speed and simplicity of this interface promotes both rapid prototyping and experimentation with the structure of the concurrent applications.

1. Introduction

Before the potential of parallel processing systems can be effectively utilized by the general programming community, a substantial retraining effort must first be undertaken. Considering the degree to which sequential architectures have become imbedded in our programming mindset, the task is not one which will be accomplished easily or overnight. The only hope of success is through a structured educational program which stresses the ideas, constructs, and theoretical foundations of parallel processing.

Studies have shown that the degree of success obtained in teaching programming is greatly influenced by the amount of hands-on exposure granted to the student. Unfortunately, such readily available exposure to parallel processing facilities is currently nonexistent. In an attempt to address this problem, the HIGHLAND system has been developed. Its purpose is to provide an accessible mechanism with which individuals who do not have convenient access to more orthodox systems can be introduced to the field of parallel processing. It accomplishes this goal by allowing a number of networked UNIX workstations, dissimilar in both vendor and performance, to be directly manipulated as a single, unified, multiprocessing environment. Its use of common communication paradigms and a graphical interface make it easy to use, highly interactive, and an ideal learning tool.

2. Previous Works

Due to editorial restrictions, a complete discussion of the previous works upon which HIGHLAND is based is not possible. A listing of the more pertinent papers, however, is provided in the bibliography.

3. System Structure

HIGHLAND is modeled after the distributed memory parallel processing model. The user is presented with a number of processing elements which are interconnected and consequently have the ability to exchange information via a set of system-supplied I/O functions. As one would expect in such an environment, parallel applications are constructed as a set of concurrently executable modules and downloaded onto some number of processing elements. Communication requirements between the processes are subsequently identified and established prior to the actual execution of the system. From this point of view, HIGHLAND appears to offer nothing new or unique, and indeed, at this level of abstraction that is exactly what was desired: a functionally generic distributed memory parallel processing environment. What makes this particular system unique is the way in which this environment is implemented and the style of interaction it offers the user.

HIGHLAND simulates both the components and the functionality of a generic distributed memory multiprocessing system using only the computational resources of a local area network. Processing elements, which are allocated and used to execute the component processes of a given application, are in fact a set of UNIX workstations. Due to the ability of the system's communication software to shield the user from various machine incompatibilities, these workstations are allowed to vary in both vendor and capability. In its current release, HIGHLAND supports a wide range of hardware platforms, including systems from such vendors as Sun

* UNIX is a registered trademark of AT&T.

0-8186-2113-3/90/0000/0742$01.00 © 1990 IEEE

Microsystems, Hewlett-Packard, Digital Equipment Corporation, Apollo, and IBM. For implementation of the interprocessor communication facility, the standard UNIX socket interface was chosen. Supported by the TCP/IP suite of network protocols and running over a standard 10 MB/second Ethernet, this transport mechanism not only offers a high degree of availability, but has also shown itself to be adequate to support larger-grain parallelism.

4. Interprocess Communication

At the program level, each user-written process is supplied by HIGHLAND with a single input port and a single output port to act as the endpoints for communication between itself and the other modules of a given application. From the module's perspective, these ports exhibit several noteworthy characteristics. First, they are strictly serial in nature, supporting no type of direct or look-ahead access. The ports are also directional, with only read operations being permitted on the standard input and only write operations being permitted on the standard output. Perhaps the most restrictive of the ports' characteristics, however, is that fact that they represent the sole mechanism by which data can enter or leave the associated process. For those who have grown accustomed to utilizing multiple input and output sources when constructing an application, this may appear to severely limit the utility of HIGHLAND's communication facilities, but such is not the case. As will be shown in a later section, this restriction is eliminated through the use of dedicated system utility processes for the implementation of more complicated data routing schemes.

The simple observations and characteristics specified above encompass the extent of a module's implicit knowledge of its I/O ports. No information is given regarding the source of the data the process is reading from its standard input, nor is any given specifying the destination of the data being written onto the standard output. Within HIGHLAND, the binding between the application's component modules, which is necessary in order to make such a determination, does not take place until the time of execution. The major advantage to this separation of process code and system configuration details is that it allows the information to be specified instead in a format more convenient than conventional text. As will be seen, this method is via the system's interactive graphical display.

5. System I/O Functions

In conjunction with the standard input and output ports, HIGHLAND also supplies a pair of system-supported communication routines through which processes can interact with them. These routines are the *hread* function, which allows a process to gather data from its standard input, and the *hwrite* function, which is used for writing data onto the standard output. Unlike some message-passing environments which supply only untyped byte transfer functions, HIGHLAND's I/O functions require messages to be both strongly and fully typed. Common scalar data types such as character, short and long integer values, as well as single and double precision floating point numbers are all supported and valid for use in the construction of interprocess messages.

In addition to supporting the transfer of messages containing one or more occurrences of a single data type, such as a string of integers or an array of floating point values, HIGHLAND also allows the construction of messages containing a composition of several distinct types. In much that same way that the C language's "struct" construct allows the collection of a set of disjoint variables for subsequent manipulation as a unit, HIGHLAND's system I/O routines offer a similar capability for message specification. By implementing its own type of structure data type, a straightforward method is offered by which any number of fields can be specified within a message while maintaining the strongly typed nature of messages of a simpler, singular type.

6. Data Translation Facilities

Since HIGHLAND was intended to operate by default in a heterogeneous workstation environment, a major concern in the design of the system's communications facilities was the automatic conversion of the various data types between machines. To accomplish this, the system-supported I/O facilities were augmented with an integrated set of data conversion routines. On output operations these routines automatically take care of interpreting the type of each value passed, a straightforward task thanks to the strongly typed nature of the message structures, and converting the data into a system independent or network data format prior to transmission. On the receiving end, corresponding utilities handle the conversion from the network format back into the local, host-specific form. The logical relationship between these elements is depicted in Figure 1.

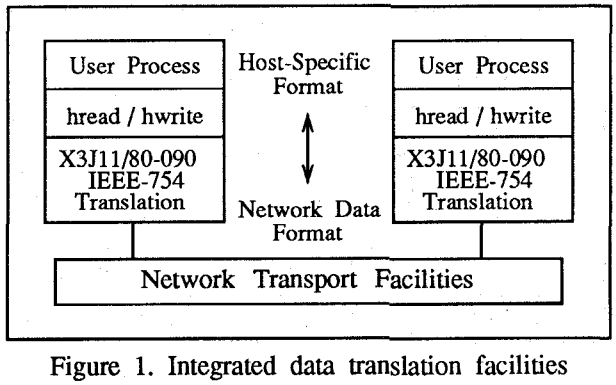

Figure 1. Integrated data translation facilities

Since the data translation routines would inflict additional overhead onto the communication process, it was strongly desirable to choose a network data format that closely reflected the most common of the various system-dependent data formats. By doing so, the effort expended in the data conversion process would be minimized for a majority of the systems used. Basing the final decision on the particular set of systems used for HIGHLAND's development, a data specification was established which in actuality is a combination of a pair of existing format standards. For the encoding of integer values, the Sun Microsystem's data representation was selected. This standard, which is formally based off of the ANSI X3J11/80-090 C language implementation standard, supports the representation of both 16- and 32-bit, signed and unsigned integer values. For representing floating point values, the IEEE-754 standard was chosen. This format provides a normalized structure for both 32-bit single precision and 64-bit double precision real values. When combined, these two standards form a comprehensive, well-established format for each HIGHLAND-supported data type.

7. System Utilities

By providing each component user process of an application with but a single input and output port, the degree of parallelism which can be achieved by the system as a whole is severely limited. At best these simple tools would allow the creation of a pipeline or a loop of concurrently-executing processes. While being extremely useful in their own right and providing sufficient process interaction to solve a number of different types of problems, these two constructs are just not applicable to all situations. In spite of the simplified interface which the scheme offers, it is obvious that a more sophisticated mechanism must be supplied and supported by HIGHLAND for the interconnection of processes and the routing of data between them. With no desire to increase the complexity of the program-level communication interface while doing so, it was decided that the best way of offering this increased functionality was to remove the more complex communication tasks from the application processes altogether and assign them instead to a set of external, system-supplied utility processes.

HIGHLAND's system utilities are not to be confused with the user processes discussed up to this point. User processes are those which are written by the programmer and comprised mainly of application-specific code. System utilities on the other hand are supplied in a ready-to-execute form and are available for use with little or no coding effort on the part of the programmer. Each utility is designed to support a specific type of routing function ranging from the very simple, such as replication and merging of data streams, to more complex functions such as automatic and program-controlled data routing. In addition, depending on the particular function implemented, each utility can maintain several input and output ports. This allows not only the off-loading of the routing logic from the user processes, but also permits the creation of communication networks of arbitrary branching factors, both fan-in and fan-out.

In addition to the savings in programmer effort gained by the centralization of these functions, another benefit is obtained. By removing these tasks both logically and physically from the component processes, the user is able to modify the data routing scheme of a parallel system at run time by merely replacing individual data routing nodes. Moreover, due to the absence of configuration information within the user code, these modifications can take place without modifying or recompiling the attached component processes.

In attempting to give a general description of the various system utilities, it would be useful to

Utility Name	Inputs	Outputs	Description
Collector	2 or 4	1	Merges multiple input streams into a single output stream.
Duplicator	1	2 or 4	Duplicates incoming messages onto each of its multiple outputs.
File Load	0	1	Creates a message stream from the contents of a specified file.
File Dump	1	0	Dumps the incoming message stream into a specified file.
Buffer	1	1	FIFO message queue.
Data Rate	1	1	Displays statistics on transfer activity between a pair of processes.
Tee-TTY	1	1	Displays contents of messages flowing between a pair of processes.
I/O-TTY	1	1	Allows the entry of messages from the console keyboard.

Figure 2. Autonomous utilities.

Utility Name	Inputs	Outputs	Control	Description
Multiplexor	2 or 4	1	Downstream	Process-controlled selection of multiple inputs.
Demultiplexor	1	2 or 4	Upstream	Process-controlled selection of multiple outputs.
Ask-for	1	2 or 4	Upstream	Demand-driven routing of outgoing messages.

Figure 3. Externally-controlled utilities.

be able to group them based on some discerning characteristic or function. Perhaps the most useful scheme for such a categorization is by the level of autonomy they exhibit over their own execution. When segregated in this way, two distinct classes of utilities can be identified. The first group, the simpler of the two, are known as *autonomous utilities*. A brief synopsis of the members of this category is given in Figure 2. As implied by the name, these utilities perform functions which are sufficiently specific and self-contained so as to require no controlling intervention by the attached processes. Once execution begins, the only operational requirement is a set of one or more locations from which the utility can retrieve its data, and a set of one or more destinations to which the data can subsequently be sent.

The second category of utilities are termed *externally-controlled utilities*. Members of this class, descriptions of which are given in Figure 3, are not nearly as self-sufficient as those of the previous group. Their functions are such that they require some degree of control be exercised over them by an immediately connected user process. Depending upon the nature of the utility, the controlling process may be situated either upstream or downstream from the utility. Since no direct, code-level link exists between them, any necessary controls are enacted by the transmission of special message types. To lessen the impact on the code of the user process, these special messages are created indirectly through calls made to built-in system functions dedicated to each type of externally-controlled utility.

8. Run-Time Environment

As shown in Figure 4, the run-time environment provided by HIGHLAND for the specification and execution of parallel applications is comprised of two distinct components. First, on each of the UNIX systems which will be utilized as a compute node, an *HServer daemon* must exist. These processes play the role of minions, permitting a certain amount of control to be exercised remotely over their respective host systems. While such facilities could constitute a source of potential security problems, care has been taken to ensure that the functionality of these processes is limited to only that required for the support of HIGHLAND. In addition, the operation of each HServer takes place using only normal user authorizations and permissions; no system or "root" level privileges are necessary. While not providing complete security, these two simple measures sufficiently limit the degree of potential damage which could be maliciously inflicted on a system.

Acting not only as the controller for the distributed HServer daemons, but as the primary user interface as well, HIGHLAND's graphical control environment constitutes the second major component of the run-time system. This process executes on the user's local machine and acts as the driving force behind a HIGHLAND session. From the user's perspective, it is this controller that creates and maintains the system's graphical display. It manages all pertinent aspects of man-machine interaction and ensures that the information shown is an accurate depiction of the current state of the application. From an overall system perspective, it is the controller that supports the illusion of a unified computing environment. It and it alone holds the knowledge of the machine dependent aspects of the underlying hardware. With this knowledge, it exercises the necessary controls over all the utilized workstations to create the illusion of a single, homogeneous multiprocessing system.

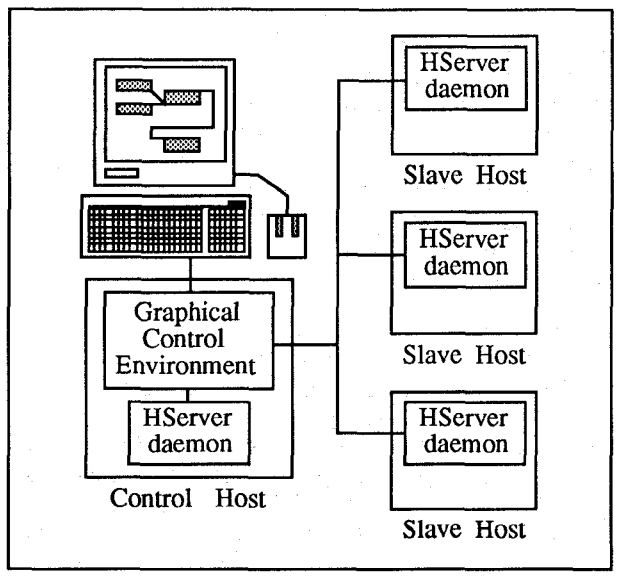

Figure 4. HIGHLAND run-time structure.

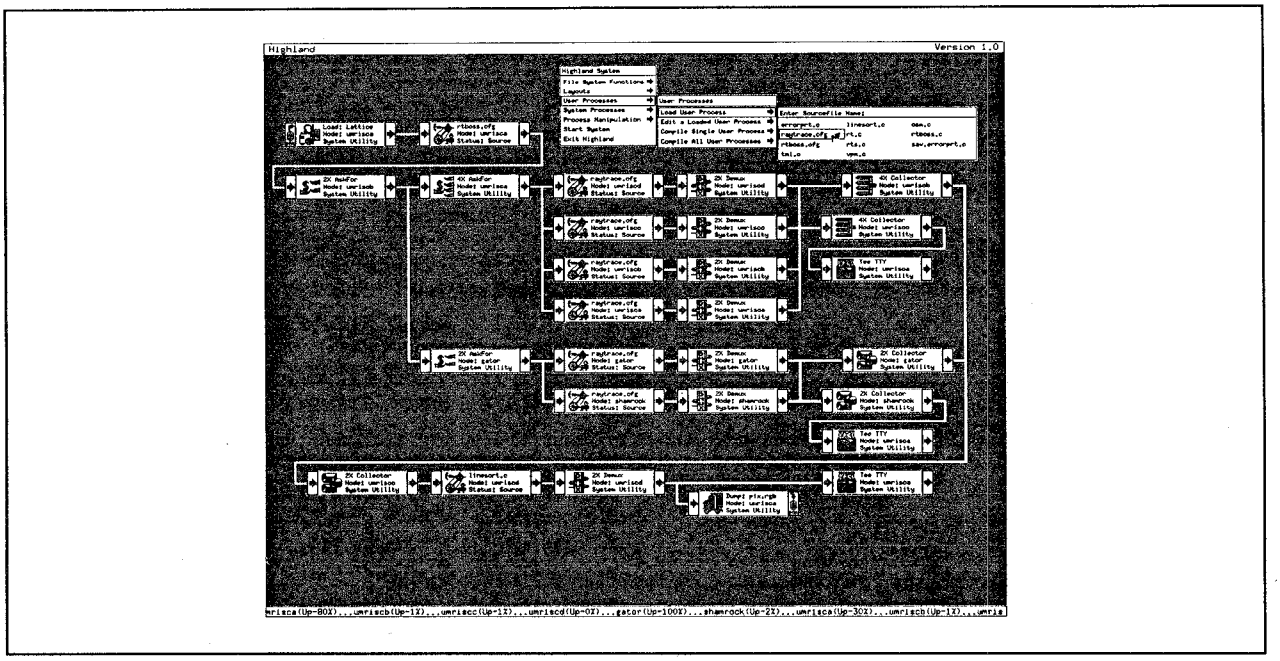

Figure 5. HIGHLAND's graphical control environment

9. Application Specification and Execution

Once an application has been designed and coded using a combination of user-written programs, system utilities, and system-supplied I/O and control functions, its formal specification to the HIGHLAND run-time environment can begin. Using the system's graphical interface (depicted in Figure 5) and guided by a series of pull-down menus, the user progresses through four distinct steps leading up to the application's execution.

Step 1: Process Load

In step one, the individual utility and user processes which will comprise the application are selected for execution. Since they exist as an integral part of the system, the selection of utility processes is straightforward. Providing the user with a complete listing of all such available processes, the menuing system allows any desired utility to be specified using only the mouse. Once selected, an iconic representation of the utility is created on HIGHLAND's graphical display through which all subsequent interaction will take place.

Due to the potential heterogeneity of the underlying hardware, user processes are introduced to HIGHLAND in source code form. In the current implementation, due mainly to the high degree of standardization it offers, only programs written in the C programming language are supported. Using the provided menu options which allow the traversal of the UNIX directory structure, the user is presented with listings of files eligible for loading into the system. From these lists, he or she may select desired processes with a click of the mouse. Once specification is complete, the process is placed onto HIGHLAND's display in icon form. All subsequent interaction with the user process will take place only through this icon.

Step 2: Link Specification

In step two, the user is requested to specify the data communication links he or she wishes HIGHLAND to establish between the currently loaded processes. Keeping in line with the desire to make the user interface as friendly and interactive as possible, this information is specified using only the mouse and the iconic representation of the component processes. The user repetitively selects pairs of process icons, in source process/destination process order, whenever a communication link is to be established between them. Then, referencing its own internal database, the system determines the validity of each requested link and provides instant feedback as to the outcome of the check. If the link was not a valid one, such as trying to connect a process which has no available ports, text windows are displayed explaining the cause of the request's rejection. If the requested link was valid, HIGHLAND immediately updates the display to reflect the instantiation of the new link.

Step 3: Parallel Compilation

In the third step the user processes, which have been loaded into HIGHLAND in source form, are readied for execution. For each, the associated source files are downloaded to the HServer daemons of their assigned hosts for remote compilation. The

compilations take place in parallel, with the compilation of all individual source files being initiated prior to any attempt being made to retrieve the executables. By doing so, the time required for the compilation of the entire parallel system is only contingent upon the longest compile time of any component user process. At the end of these parallel compiles, as is the case in any compilation, there are two possible outcomes. If either syntactic or linkage errors are discovered, a log of the errors is returned for use in subsequent debugging. If the compilation completes successfully, the executable version of the process is returned to the controlling host where it is stored until the time of execution.

Step 4: Execution and Control

In the fourth and final step, the parallel application is initiated. In what, from the system level, is by far the most complicated of the four steps, HIGHLAND downloads the now executable processes to their target systems, automatically establishes the specified communication links over the network socket interface, and starts the execution of the system. The details of this process, however, are hidden entirely. From the user's perspective, outside of a simple text window which describes the current state of the start-up process, this phase appear no more or less complex that those previously discussed.

Once execution of the parallel program has begun, HIGHLAND's graphical interface ceases being a mechanism for constructing applications and becomes instead a means of controlling them. From within the display, a number of powerful capabilities are provided which allow the user to exercise complete authority over the executing parallel system. A real time display of remote workstation utilizations is supplied, providing a method of gauging the effective parallelism of the application over time. At a more microscopic level, tools also exist which allow individual link traffic to be measured and monitored. Through their use it is possible to pinpoint potential bottlenecks in the system's overall dataflow. When problems or inefficiencies such as these are encountered, it is possible to abort individual processes as well as cancel the execution of the application entirely. This, however, is not to be considered a loss of all work done up until this point.

Due to the independence of the component processes and the ability of the HIGHLAND system to control them, it is possible to reconfigure around potential problems without the need of starting the entire construction process from scratch. Nodes can be added, deleted, or reassigned to different host processors. Likewise, additional communication links can be requested and existing links can be removed or rearranged. Upon the completion of any reconfiguration, HIGHLAND ensures that only the minimal amount of work is performed to get the overall system back to an executable status. With such minimization, the overall cycle time between successive configuration attempts is very small; a fact which encourages experimentation with the structure of the parallel application.

10. Conclusion

HIGHLAND has been successfully ported and used across several types of workstations. Utilizing these systems, a number of applications have been developed and several more are currently in progress. Based on experiences gathered to date, commonly available LAN resources have proven themselves sufficient for the support of larger-grained parallel processing applications. The future of the HIGHLAND system looks very promising.

Bibliography

[1] Baily, M.L., Socha, D., and Notkin D., "Debugging Parallel Programs using Graphical Views", *Proceedings of the 1988 International Conference on Parallel Processing*, 1988, Vol. 2, pp. 46-49

[2] Boarder, J.C., "Graphical Programming for Parallel Processing Systems", *Proceedings of the Second International Conference on Distributed Computing Systems*, 1981, pp. 467-475

[3] Geigel, T., and Pagan, M., "A Distributed Application of the PHARROS Project", *Proceedings of the 1988 International Conference on Parallel Processing*, 1988, Vol. 2, pp. 110-113

[4] LeBlanc, R.J., and Robbins, A.D., "Event-Driven Monitor of Distributed Programs", *Proceedings of the 5th International Conference on Distributed Computing Systems*, 1985, pp. 515-522

[5] Nadas, T., and Fournier, A., "GRAPE: An Environment to Build Display Processes", *Computer Graphics*, July 1987, Vol. 21, No. 4, pp. 75-83

[6] Nichols, K.M., and Edmark, J.T., "Modeling Multicomputer Systems with PARET", *IEEE Computer*, May 1988, Vol. 21, No. 5, pp. 39-48

[7] Snyder, L., "Parallel Programming and the Poker Programming Environment", *IEEE Computer*, July, 1984, Vol. 17, No. 7, pp. 27-36

[8] Stotts, P.D., "The PFG Language: Visual Programming for Concurrent Computation", *Proceedings of the 1988 International Conference on Parallel Processing*, 1988, Vol. 2, pp. 72-79