Dynamic task distribution in a heterogeneous loosely-coupled distributed computer system

Samuel K West

University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/522

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Dynamic Task Distribution in a Heterogeneous Loosely-Coupled Distributed Computer System

by

Samuel K. West

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
August 1995
The Thesis of Samuel K. West for the degree of Master of Science in Computer Science is approved.

Chairperson, Kia Makki, Ph.D.

Examining Committee Member, John Minor, Ph.D.

Examining Committee Member, Angelo Yfantis, Ph.D.

Graduate Faculty Representative, Woosoon Yim, Ph.D.

Interim Dean of the Graduate College, Cheryl L. Bowles, Ed.D.

University of Nevada, Las Vegas
August 1995
Abstract

This thesis studies the problem of dynamic distribution of tasks between hosts in a heterogeneous, loosely-coupled, distributed computing system. The goals of the study are to (a) demonstrate reduced execution time in a computer program making subroutine calls to be executed on a computer (or computers) which will yield better performance than the one on which the program was initiated, (b) demonstrate the feasibility of dynamic task-to-host binding, (c) demonstrate the feasibility of a programmer-transparent methodology of distributed computing using a library approach. These goals are partially realized using the Remote Procedure Call protocol in a programmer-transparent framework of library calls. Examples of a distributed library, libHCS, and an associated daemon, HCSdaemon, implemented in support of these goals, are analyzed for their feasibility and effectiveness in solving this problem. Although results of the study fail to demonstrate reduced execution time, dynamic task-to-host binding and programmer transparency were achieved. Further study is indicated.
Acknowledgments

I would like to thank my advisor, Dr. Kia Makki, for his patience and encouragement. I would also like to thank Dr. Baharam Nassersharif, director of the National Supercomputing Center for Energy and the Environment at UNLV, for the use of the Center's hardware and software. Finally, thanks to Karen, Michael and Tiffin, for everything else.
# Table of Contents

Abstract ......................................................................................................................................... iii  
Acknowledgments ......................................................................................................................... iv  
Table of Contents ......................................................................................................................... v  
List of Figures ............................................................................................................................ viii  
List of Tables .................................................................................................................................. x  
Chapter 1 Introduction .......................................................................................................... 1  
Chapter 2 Terms and Notation .......................................................................................... 7  
  2.1 Terms ........................................................................................................................ 7  
  2.2 Notation .................................................................................................................... 9  
Chapter 3 Related Research and Existing Systems ..................................................... 10  
  3.1 NQS ......................................................................................................................... 10  
  3.2 NQE ......................................................................................................................... 12  
  3.3 DCE ......................................................................................................................... 12  
  3.4 Linda ....................................................................................................................... 13  
  3.5 PVM ........................................................................................................................ 14  
  3.6 MPI .......................................................................................................................... 15  
  3.7 HeNCE .................................................................................................................... 16  
  3.8 DHSMS .................................................................................................................. 16  
  3.9 Jade ......................................................................................................................... 17  
Chapter 4 Abstract Model ................................................................................................. 18  
Chapter 5 Computational Model ...................................................................................... 24
Appendix C  HCSdaemon......................................................... 111
Appendix D  Simulator ....................................................... 136
Bibliography ................................................................. 160
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>NQS Model of Distributed Computing</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>RPC Model of Distributed Computing</td>
<td>13</td>
</tr>
<tr>
<td>3.3</td>
<td>PVM Model of Distributed Computing</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Algorithm MatrixMultiply</td>
<td>22</td>
</tr>
<tr>
<td>5.1</td>
<td>Pseudocode for choose_host()</td>
<td>25</td>
</tr>
<tr>
<td>5.2</td>
<td>Pseudocode for complexity()</td>
<td>26</td>
</tr>
<tr>
<td>5.3</td>
<td>Pseudocode for task()</td>
<td>27</td>
</tr>
<tr>
<td>5.4</td>
<td>Conventional Subroutine Model</td>
<td>28</td>
</tr>
<tr>
<td>5.5</td>
<td>Separately Compiled Library</td>
<td>29</td>
</tr>
<tr>
<td>5.6</td>
<td>libHCS Model of Distributed Computing (simplified)</td>
<td>30</td>
</tr>
<tr>
<td>6.1</td>
<td>RPC Thread of Control</td>
<td>32</td>
</tr>
<tr>
<td>6.2</td>
<td>libHCS Model of Distributed Computing</td>
<td>34</td>
</tr>
<tr>
<td>6.3</td>
<td>libHCS Thread of Control</td>
<td>35</td>
</tr>
<tr>
<td>6.4</td>
<td>Typical libHCS Subroutine Stub</td>
<td>36</td>
</tr>
<tr>
<td>6.5</td>
<td>Pseudocode for choose_host()</td>
<td>37</td>
</tr>
<tr>
<td>6.6</td>
<td>Compiling an rpcgen(l) Protocol File</td>
<td>39</td>
</tr>
<tr>
<td>6.7</td>
<td>Building libHCS and HCSdaemon</td>
<td>40</td>
</tr>
<tr>
<td>6.8</td>
<td>Linking a User’s Program</td>
<td>41</td>
</tr>
<tr>
<td>6.9</td>
<td>Sample User Program</td>
<td>42</td>
</tr>
<tr>
<td>6.10</td>
<td>Subroutine addf() in libHCS.c</td>
<td>43</td>
</tr>
<tr>
<td>6.11</td>
<td>addf_1() from libHCS_clnt.c</td>
<td>43</td>
</tr>
<tr>
<td>6.12</td>
<td>addf() 's XDR Routines</td>
<td>44</td>
</tr>
<tr>
<td>6.13</td>
<td>addf() 's Relevant Entries in libHCS_svc.c</td>
<td>45</td>
</tr>
<tr>
<td>6.14</td>
<td>addf_1() in libHCS_1.c</td>
<td>46</td>
</tr>
</tbody>
</table>
List of Tables

Table 7.1: Hosts in the Distributed Computer ........................................ 53
Table 7.2: Candidate Routines in clinpack ............................................ 55
Table 7.3: Additional Host in the Distributed Computer ......................... 56
Table 8.1: Benchmark Execution Matrix ............................................ 62
Table 8.2: Subroutine Complexity and Benchmark Size ......................... 62
Table 8.3: Dynamic Parameters of the Simulation .................................. 63
Table 8.4: choose_host() Policy Elapsed Times .................................... 67
Table 8.5: Random Policy Elapsed Times ............................................. 68
Table 8.6: Least-loaded Policy Elapsed Times ....................................... 69
Table 8.7: Strongest Policy Elapsed Times ........................................... 70
Table 8.8: All Policies Elapsed Times ................................................ 71
CHAPTER 1

Introduction

Distributed computing is a widely studied topic. Many of the studies have resulted in production systems. A subset of these studies seeks to solve the general problem of distributing computing tasks across a collection of computers to allow the entire collection\(^1\) to cooperate in a single, global computation. These studies have varied, as will be described in the following paragraphs, according to their use of homogeneous vs. heterogeneous computers, static vs. dynamic configurations, loose vs. close-coupling, and coarse vs. fine granularity of distributable tasks.

Some of the studies mentioned have utilized computers which were of the same type (homogeneous), running the same operating system, and connected via a single local-area network, as is the case with a cluster of workstations. While other studies have focused on computers of varying types (heterogeneous), running different operating systems or different versions of the same operating system, and connected via a wide-area network like the Internet [7,2]. In terms of high performance computing, it should be noted that, as the number of specialized architectures continues to increase, any collection of computers intended to be used as a distributed computer will likely be heterogeneous and consist of one or more representatives from several of these

\(^1\) Referred to in this thesis as a *distributed computer*. 
architectures. The reason for assembling such a collection would be to improve program performance by matching the intrinsic computational requirements of arbitrary algorithms with available and suitable hardware to run those algorithms. Since most programs do, in fact, exhibit a variety of computational types\(^2\), a corresponding variety of architectures is needed to extract maximum performance from any given program\(^{[15]}\).

In some of these studies the configuration of the distributed computer has been inflexible, static, and tightly-coupled. These systems require the programmer to design in advance the configuration of the distributed computer, and are incapable of allowing host additions and deletions or of surviving host failure\(^{[24,25]}\). On the other hand, some studies allow the configuration to be robust, dynamic and loosely-coupled, capable of surviving host failure, or even host additions and deletions\(^{[7]}\). Likewise, in these various studies, the tasks to be distributed range from the coarsest granularity, like entire programs or jobs, to very fine granularity, like subroutines or individual instructions.

The stated goals of these studies are similarly varied. Sometimes the goal has been to balance the computational load across the distributed computer in an attempt to achieve the highest possible throughput of the total system by insuring that no process has to wait for an oversubscribed resource on one computer even though some member of the system might be idle. Sometimes the goal has been to achieve the maximum performance of a single job, under the assumption that a large, expensive collection of hardware ought to be usable in a way that makes the resultant distributed computer equal, at least, to the sum of its parts. And sometimes the goal has been to simply find a way to use idle computing power that might be available from, say, a collection of

---

\(^2\) e.g. parallel vs. sequential, vector vs. scalar.
workstations that go largely unused after working hours and on weekends.

Finally, the production systems resulting from these studies take many forms. These range from whole job distribution systems like NQS, the Network Queueing System, in which the requirements are fairly simple - deliver a program and its input data to a specific remote host, where it will be executed, and return the output to the local host (frequently in a homogeneous collection of computers), to parallel programming on tightly-coupled architectures like the CM-5, to coordinating interprocess communication in networks of homogeneous computers with Linda [4], to superconcurrency in a heterogeneous suite of processors[16,17], and even instruction-level distribution on very tightly-coupled, specialized architectures like the IBM 360/91.

The present thesis proposes a solution to the following specific problem: Create a programmer-transparent environment for the distributed execution of a program within a dynamically changing, heterogeneous, loosely-coupled distributed computer system. The goal of the solution will be to improve execution time of the program by, potentially, distributing the elemental tasks comprising the program to computers in the distributed system better suited to execute them than the computer on which the program was invoked. The choice of host to which to bind each task will be made at execution time of the task according to a heuristic scheduling algorithm which will seek to minimize the expected elapsed time for execution of the task by considering the current host and network load characteristics of the distributed computer and the intrinsic performance match, as evidenced by benchmark results, between the task and each candidate host.

A program executing within this system is assumed to make calls to one or more members of a set of subroutines contained in a special library. These subroutines will be
the basic units of task distribution. To satisfy programmer-transparency the solution will require only a conventional compiler and loader, conventional sequential programming model, and an ordinary memory model. Further, no specialized knowledge of interprocessor communication, or interprocess communication for that matter, will be demanded of the programmer. It will only be required that the programmer code his or her program using one or more of the subroutines contained in the aforementioned library. In previous solutions to this problem, techniques above and beyond the simple compile, load and run were needed to take advantage of the additional computing power available in a distributed computer. This solution seeks to obviate those needs, placing the burden at a lower level than that where the applications programmer ordinarily works.

The present solution involves, first of all, the creation of the new library, libHCS, which will be interposed between the loader and the system libraries on the local host. This library will contain subroutine stubs which correspond by name to the subroutines to which the programmer makes calls in his program. This means that, when the programmer links the program, the loader will resolve the distributable subroutine calls to the stub entries in libHCS instead of to the entries in the system libraries. libHCS will also contain support routines to manage communication between the program and the distributed computer, while the standard subroutine stubs will take care of making the proper request to the distributed computer and coordinating the transfer and possible conversion of parameter lists and return values.

The rest of the distributed system will be implemented by a daemon, HCSdaemon, which will be converted to run on each participating computer in the collection. The local library, libHCS, will accept requests for compute service (in the
form of subroutine calls) from user programs with which it has been linked. Once a call is made from the user program the local library will attempt to discover, by communicating with the HCSdaemons running on the various computers in the distributed system, the best candidate in the distributed computer able to execute the request. This will require that each HCSdaemon on each host have knowledge of the characteristics of the machine on which it is running vis-à-vis the characteristics of the various distributable subroutines. Specifically, each HCSdaemon will have access to the performance characteristics of the particular routine on its host in the form of benchmark data for the routine. Likewise, to make an informed choice of candidates, the current load of each machine must be made available to the local library. The HCSdaemons must provide this information in some form.

Having identified the best candidate, the library will transfer the subroutine ID and its arguments to the servicing computer, where the requested subroutine will be executed. Following subroutine execution, the remote host will return the results. The local library will then complete the circle by returning the results to the calling program.

It should be clear from the foregoing that there are many issues which must be addressed in order to achieve the proposed solution. Some of these issues are:

- Data portability and uniform parameter passing
- Library and daemon portability
- Subroutine conversion and library extensibility
- Interprocess and interprocessor communication and synchronization
- Global state of the distributed computer
- Stateless vs. stateful daemon operation
- Metrics for determining host suitability
- Pathological couplings and global data
All of these issues will be considered, with possibilities for solution and implementation proposed and explored.

Motivation for this type of work has been well documented [19]. The present study is motivated by the need for a simple means to achieve cost-effective Heterogeneous Computing, without the usual requirement of development and optimization. Also, a significant question answered by this study is whether it is feasible to make the sort of scheduling choices and dynamic task-to-host binding employed by libHCS. As already mentioned, numerous research projects in this area have been turned directly into production systems [4,5,18,20]. A number of these projects will be described in Chapter 3, Related Research and Existing Systems. This is an important area and will continue to be studied in the future.

The organization of the balance of this thesis follows. Chapters 2 and 3 contain some terms and notation, and related research and existing systems. Chapter 4 is a specification of the abstract model, followed by the corresponding computational model in Chapter 5. Chapter 6 contains the initial implementation. This implementation should point the way to future work to be done in this area as well as addressing the foregoing issues and answering some of the questions raised by the computational model. Chapter 7 describes the prototype and the first attempt to extend the system by adding new subroutines in an implementation case study. In Chapter 8 the scheduling algorithm which achieves host selection, is simulated in program execution over a large number of randomized network loads to experimentally demonstrate its viability. The final chapter draws some conclusions, suggests future research, and offers some final remarks.
CHAPTER 2

Terms and Notation

2.1 Terms

HCS. Heterogeneous Computer System. A collection of computers of arbitrary architectural types (e.g. Massively Parallel, Vector, SuperScalar, etc.) connected via a communications network. Implicit, at least for the purposes of this thesis, is the notion that this collection of computers should be capable of some level of cooperation toward the fulfillment of some overarching goal (see HC, below.)

HC. Heterogeneous Computing. What Heterogeneous Computer Systems do. The use of a loosely-coupled, heterogeneous suite of computers toward the fulfillment of some overarching goal.

HPC. High Performance Computing.

Superconcurrency. A general technique for matching and managing optimally configured suites of super-speed processors [16,17]. So-called because it is an approach to supercomputing and because it concurrently uses concurrent (vector and parallel) processors. Ideally, the sequential programming concepts presented in this thesis could be extended to the parallel realm.

IPC. InterProcess (and/or InterProcessor) Communication. Any mechanism which
allows two, concurrently executing processes, either on the same host or across a network, to communicate instructions or data. Common mechanisms include: Shared memory with semaphores; and Message Passing.

ONC. Open Network Computing. The portable, distributed computing platform developed by Sun Microsystems. It consists of the RPC (Remote Procedure Call) routines and the XDR (eXternal Data Representation) routines contained within the RPC Library.

Network. Specifically: computer network. A collection of computer systems able to communicate with one another over any of a variety of media, using an agreed upon protocol. For the purposes of this thesis, IP (Internet Protocol) will be the communications protocol and the media will be unspecified.

Task. A separately identifiable execution of a computer algorithm. For the purposes of this thesis, task will be synonymous with the common notion of subroutine.

Loosely-coupled computer system. A computer system with a dynamic configuration in which the individual processing elements are physically separated, perhaps widely separated. Processing elements may communicate via a network to solve a single, global computation. No common clock exists for synchronization, which must be managed by software mechanisms executed by the elements themselves. Processing elements are allowed to fail or not participate without affecting the correctness of the global computation.

Tightly-coupled computer system. A computer system in which the individual processing elements are located in close proximity (usually in the same box) to each
other. They share a common clock and/or memory which may be used for synchronization of computational tasks.

2.2 Notation

Pseudocode and computer code will be represented with courier font. In circumstances where the meaning of a pseudocode is apparent from context, considerable license will be taken.

The implementation described in this thesis was done on a network of computers which all run variations of the UNIX operating system. This is not a requirement, but there is a distinct UNIX bent to most of the presentation.

On UNIX systems, a library, such as the JUNK library would ordinarily reside in a file named libJUNK.a. But, in this thesis libHCS will be used to refer to the library residing in file libHCS.a.

Filename extensions:

- '.a' refers to UNIX relocatable library files.
- '.x' refers to rpcgen protocol files.
- '.c' refers to C language source files.
- '.h' refers to C language header files.
CHAPTER 3

Related Research and Existing Systems

The following is a survey of existing production systems, as well as several research projects under way. This section illustrates the variety of approaches that have been taken to solve the problems of coordinating multiple computer systems to the performance of computing tasks. In all cases, techniques beyond simple compile and load are needed to achieve any level of task distribution. This sets apart all these approaches to distributed computing from the present investigation. Also, these systems are not generally self-optimizing, a central feature of the present system. Further, the libHCS/HCSdaemon system is a) heterogeneous, because HCSdaemon may be easily ported to almost any architecture which supports RPCs, b) dynamic, since participating hosts may be added or removed at any time, c) loosely-coupled, since RPCs will even operate over a network as widely dispersed as the Internet, and d) fine-grained, in that subroutines are the basic units of task distribution. Where appropriate, the following systems have been similarly classified according to their heterogeneity, flexibility, coupling and task granularity.

3.1 NQS

One of the simplest solutions to the problem, whole-job distribution, is the Network Queuing System[26]. NQS is a public-domain job submission and monitoring
system which, like many systems of similar type, assumes a static configuration of available systems and requires the programmer to specify the particular machine on which his or her job will run. There is little portability available in this system, in that

![Diagram of NQS Model of Distributed Computing]

Figure 3.1: NQS Model of Distributed Computing
programs which will be run on any given system in the NQS-managed network are required to be compiled for, or on, the target system. NQS takes care of delivering the job script to the proper computer, causing it to be executed at the proper time, and returning output files and job status to the computer from which the job was submitted (see Figure 3.1). NQS is heterogeneous, static, loosely-coupled and coarse-grained.

3.2 NQE

Cray Research, Inc.'s Network Queuing Environment [27] provides an integrated computing environment with high performance and reliability. Automatic destination selection across heterogeneous networks enables users to select the machine(s) on which their jobs will run, allowing users to take advantage of computing resources that best meet their needs. Load-leveling automatically selects the least loaded system on which to run a job, assuring users and network administrators that work is more evenly distributed to available resources. Destination selection and load-leveling make it easy for users to do their jobs right. They just submit their jobs to the network and let NQE determine the best way to get it done. NQE is heterogeneous, dynamic, loosely-coupled, coarse-grained.

3.3 DCE

OSF's DCE is layered software that resides between computers' operating systems and an application program. The software masks the physical complexity of the multi-vendor networked environment by enabling applications to be automatically segmented and executed on the system best suited for processing each segment. OSF's full suite of DCE software has two components: Core Services and Extended Services. Core Services include features like Remote Procedure Call (RPC) (see Figure 3.2), which distributes application execution, and Time Service, which synchronizes, within
the obvious theoretical limitations, all computer clocks on the network. Extended Service includes the Distributed File System (DSF), which gives users transparent access to files stored on remote systems, regardless of their geographic location. DCE is heterogeneous,

Figure 3.2: RPC Model of Distributed Computing

dynamic, loosely-coupled, fine-grained.

3.4 Linda

Linda [4,20] is a "coordination language". A coordination language provides
operations for process creation and inter-process communication. A coordination language mated with a conventional, serial programming language yields a general-purpose concurrent-programming environment. It was originally implemented in a homogeneous environment, but was extended to support limited heterogeneity (Network Linda.) Linda is the product of Scientific Computing Associates, Inc. and runs on shared-memory parallel computers, on distributed memory computers, and on local area networks (e.g. Unix workstations.)

The central issue in Linda is the loosely-coupled character of communication. Linda processes communicate indirectly via a so-called tuple space with persistent objects which are the tuples. This contrasts to more conventional communication strategies which involve direct communication via message passing or procedure calls.

Linda is homogeneous, static, loosely-coupled, fine-grained.

3.5 PVM

The Parallel Virtual Machine [25] was developed at Oak Ridge National Laboratory and is a software system that enables a collection of heterogeneous computer systems to be used as a coherent and flexible concurrent computation resource. The individual machines may be shared- or local-memory multiprocessors, vector supercomputers, specialized graphics engines, or scalar workstations that may be interconnected by a variety of networks.

PVM support software executes on each machine in a user-configurable pool and presents a unified, general, and powerful computational environment for concurrent applications. User programs, written in C or Fortran programming languages, gain access to PVM in the form of library routines for functions such as process initiation,
message transmission and reception, and synchronization via barriers or rendezvous.

Executable subroutines are ported to machines in the network which will participate in the computation (see Figure 3.3). PVM is heterogeneous, static, loosely-coupled, fine-grained.

3.6 MPI

The Message-Passing Interface[24] is an emerging standard for the specification of portable message-passing libraries. Similar in scope and purpose to PVM and authored by the MPI Forum, it is a collaborative effort whose stated goal is to "... develop a widely used standard for writing message-passing programs. As such the
interface should establish a practical, portable, efficient, and flexible standard for message passing”[24]. MPI is heterogeneous, static, loosely-coupled, fine-grained.

3.7 HeNCE

The Heterogeneous Network Computing Environment [25], likewise developed at ORNL, is a graphical interface tool and methodology for using PVM. HeNCE permits the specification of applications using a variant of directed acyclic graphs; individual nodes are associated with application modules and executed under PVM.

During or after execution, HeNCE displays an event-ordered animation of application execution, enabling the user to visualize relative computational speeds, processor utilization, load imbalances, and message traffic. A separate display shows the allocation of modules to PVM host machines. HeNCE also supports the graphical configuration of PVM hosts, assists in the generation of architecture-dependent object modules, and contains provisions for task scheduling based on user-supplied cost matrices.

3.8 DHSMS

The Distributed Heterogeneous Supercomputing Management System [14] is an integrated approach to Distributed Heterogeneous Supercomputing System management which allows management of both computational and network resources by adapting to application needs and providing a true superconcurrent environment.

The DHSMS includes a systematic methodology for both code profiling and analytical benchmarking. A Universal Set of Codes (USC) generates architecture-dependent code profiles at varying levels of detail. DHSMS takes account of both I/O benchmarking and network interface delay using a cache of network data to increase
performance.

Using a generalized and precise method, applications are characterized not only by their "degree of suitability" to a specific machine, but also by communication interaction characteristics, since data must be exchanged among machines that may have diverse I/O architectures as well as network interfaces with drastically different performance profiles.

3.9 Jade

Jade [5] is a high-level, implicitly parallel language designed for exploiting coarse-grain, task-level concurrency in both homogenous and heterogeneous environments. Jade presents the programmer with the dual abstractions of a single address space and serial semantics. Instead of using explicitly parallel constructs to create and synchronize parallel tasks, Jade programmers guide the parallelization process by providing the high-level, application-specific information required to execute the program in parallel on a heterogeneous collection of machines. With Jade, the programmer must specify three things: (1) a decomposition of the data into the atomic units the program will access, (2) a decomposition of the sequential program into tasks, and (3) a description of how each task will access data. Given this information, the implementation automatically extracts and exploits the task-level concurrency present in the computation. Jade is heterogeneous, static, loosely-coupled, coarse-grained.
CHAPTER 4

Abstract Model

In practical terms, it is the goal of this thesis to describe the development and implementation of a model of distributed computing which uses an opportunistic scheduling algorithm for choosing a host within a distributed computer to execute a particular task which will minimize the elapsed time we can expect to achieve given the state of the distributed computer (participating hosts, network latencies, machine loads and performance characteristics) and qualities of the current instance of the task (problem and dataset size.) The following rules, among others, will be observed by the algorithm: When a task is ready for execution, don’t wait to make a choice of host; and once the choice is made, send the task and execute it to completion.

This model and its implementation will differ from previous work in, at least, the following ways:

- Task to Host binding is performed at run-time based on load and other factors.
- Host additions and deletions are allowed at any time.
- The distributed nature of the system is completely transparent to the programmer and requires only a conventional compiler and loader.

This work is motivated by the need for a simple means to achieve cost-effective Heterogeneous Computing, without the usual requirement of development and
optimization.

In abstract terms we would like to describe an *elapsed time* function which will provide the basis for making the aforementioned host choice. If we define a program $P$ to be a sequence of tasks which must be executed in order, and make the assumption that the loading characteristics of the distributed computer are independent of the execution of program $P$, then the orderly application of our function to each task in the program should result in the sequence of hosts which will assure the lowest elapsed time for the entire program that we can reasonably expect to achieve.

Clearly, we cannot know, *a priori*, the choice of host which will guarantee the lowest elapsed time for task execution since host and network loads are dynamic. Rather than allow this to paralyze our ability to choose, we will use our elapsed time function to take our best guess, hoping that the loading characteristics of the distributed computer will change only slightly during execution of the task.

In a static network of computer systems, $H$, with a fixed set of tasks, $T$, it would be a simple thing to make the sort of choice just described. For example, assume a static network consisting of $n$ hosts, $H = \{h_1, h_2, h_3, \ldots, h_n\}$, and a collection of $m$ tasks, $T = \{t_1, t_2, t_3, \ldots, t_m\}$. Likewise, assume that there exists a set of $n$ network connections, $C = \{c_1, c_2, c_3, \ldots, c_n\}$, linking the primary host with each of the distributed hosts and datasets, $D = \{d_1, d_2, d_3, \ldots, d_m\}$, consisting of the data which must be transferred for each of the $m$ tasks to execute as well as the results of execution. The cross products of these two pairs of $m$ and $n$-tuples, $T \times H$ and $D \times C$, suggests new quantities, $e_{ij}$ and $x_{ij}$, where $e_{ij}$ is time for execution of task $i$ on host $j$, and $x_{ij}$ is the transfer time for dataset $i$ over network connection $j$. $e_{ij} + x_{ij}$ will be taken to denote
elapsed time of task $i$ on host $j$.

Using this notation the correct choice to execute task $i$ would be the host $j$ which minimizes the following expression

$$\min_{j = 1, n} \left( e_{ij} + x_{ij} \right)^1.$$ \hspace{1cm} (4.1)

Of course it is unrealistic, not to mention uninteresting, to assume a static network of computers. Loads, both network and processor, are continually changing as users enter and exit the system. If we allow for a dynamic network, then we must adjust our equation to account for the passage of time. Now, instead of simple $x_{ij}$ values, we have $x_{ij}(t)$, where $x_{ij}$ is a function of time. Our choice of host, $j$, now depends on time $t$, and becomes

$$\min_{j = 1, n} \left( e_{ij} + x_{ij}(t) \right)^1.$$ \hspace{1cm} (4.2)

Similarly, allowing for changing loads leads to

$$\min_{j = 1, n} \left( e_{ij}(t) + x_{ij}(t) \right)^1.$$ \hspace{1cm} (4.3)

Is expression 4.3 of any practical value? To answer that we must first define what is really meant by $e_{ij}(t)$ and $x_{ij}(t)$. Assume the existence of functions $\lambda_j(t)$ and $\tau_j(t)$ which, respectively, define the load (as a percent of availability) on host $j$ at time $t$ and the traffic (as a percent of saturation) to connection $j$ at time $t$. Further, assume that if the execution time for task $i$ on host $j$ is $e_{ij}$ when host $j$ is 100% available, then the execution time will be $e_{ij}/\lambda_j(t)$ when host $j$ is $\lambda_j(t)$% available. The same can be said for transfer time, although the notion of network saturation is a little more slippery. These assumptions lead to the following two equations

\[ \text{1 The reason for separating network transfer time from the task execution time will become clear in a moment.} \]
which are, of course, naive, and assume a linear relationships between execution time and load, and transfer time and traffic; however, these relationships will be used without further justification. These equations also suggest that load and traffic, which have just been converted in our model to dynamic entities, remain fixed for the duration of the execution of task \( j \) and the transfer of dataset \( j \). In fact, it is expected that load and traffic will change during execution. In Chapter 8, we will use Monte Carlo methods to simulate this system under dynamic loads to test the hypothesis that good scheduling choices may be made even in the presence of unpredictable changes in load. Although an interesting variant would be to consider load trends, this will not be considered in this thesis. Combining 4.4 and 4.5 with equation 4.3 yields the following expression

\[
\min_{j=1,k} \left( \frac{e_{ij}}{\lambda_j(t)} + \frac{x_{ij}}{\tau_j(t)} \right).
\]

Now let's go back and revisit \( e_{ij} \). We have defined this to mean 'the execution time of task \( i \) on host \( j \)'. This needs to be refined. Indeed, the notion of task itself needs further definition. For instance, if task \( i \) were 'multiply matrix \( A \) by matrix \( B \) and return the product', do we mean only a specific instance of \( A \) and \( B \)? Up until now, that is exactly what we meant. However, this is not a very useful meaning. It would require that, if we were to actually use equation 4.4, we would need to know \( e_{ij} \) for each and every one of an arbitrary number, say \( k \), of instances of task \( i \) on host \( j \). Of course, this is not very reasonable. What we really mean when we define task \( i \) as 'multiply matrix \( A \)

\[2\] Note that the use of \( k \) is not meant to imply that this is an enumerable set.
by matrix $B$ and return the result’ is ‘apply the algorithm, $\text{MatrixMultiply}$, to an instance of matrices $A$ and $B$ (one of $k$ possibilities) and return the result’. So, more generally, task $i$ is ‘apply algorithm $i$ to some dataset $k$’, and our definition becomes $e_{ij} = a_{ij}(d_k)$.

Does this get us anywhere? Not really. We’ve just traded ‘$k$ different instances of task $i$ on host $j$’ for ‘apply $a_i$ to $k$ datasets on host $j$’. In either case to achieve $e_{ij}$ we would need to know the execution time of every instance of $i$ on $j$.

What we would like is to be able to represent $e_{ij}$ with a derived value. We can do this if we know something about the nature of task $i$, or, more specifically, algorithm $i$. To continue with our previous example, take the $\text{MatrixMultiply}$ algorithm shown in

\begin{verbatim}
MatrixMultiply(A,m,o,B,o,n)
    /*
     * Perform the matrix multiply algorithm on m by o matrix A, and o by n matrix B and
     * return the m by n matrix C as the result.
     */
    for i = 1 to m
        for j = 1 to n
            C(i,j) = 0
            for k = 1 to o
                C(i,j) = C(i,j) + (A(i,k) \* B(k,j))
    return(C)
\end{verbatim}

\textbf{Figure 4.1: Algorithm MatrixMultiply}

This algorithm executes in $mno$ steps. Let’s say that we know its execution time, $e'$, on a specific dedicated host for an instance of $A$ and $B$, say $A [m', o']$ and $B [o', n']$. Using this knowledge, we can predict its dedicated execution time on that host for an arbitrary instance of $A$ and $B$, say $A[m_1, o_1]$ and $B[o_1, n_1]$. Now, instead of $m'n'o'$ steps, we have $m_1n_1o_1$ steps and its execution time will be $e' \frac{(m_1n_1o_1)}{m'n'o'}$. In other words, we
would like to say that the product of the execution time of a known instance of a task, and the ratio of the complexity of an unknown instance of the task to the complexity of the known instance of the task should yield the execution time of the unknown instance of the task. We will use this relationship without further justification. Intuitively, it would seem that the quality of the predictions we can make with this technique is dependent on the accuracy with which we estimate the complexity of the task. To continue, if we represent the complexity function of task \( i \) on dataset \( k \) as \( E_i(d_{ik}) \) and abbreviate the complexity of the known instance of task \( i \) to \( E_i \) and the execution time of the known instance of the task as \( e_{ij} \), then we can write

\[
e_{ij}(d_{ik}) = e_{ij} \frac{E_i(d_{ik})}{E_i}. \tag{4.7}
\]

Extending this notation to our data transfer expression gives

\[
x_{ij}(d_{ik}) = x_{ij} \frac{X_i(d_{ik})}{X_i}. \tag{4.8}
\]

where function \( X(d) \) is simply the size of dataset \( d \). Substituting these two expressions in 4.6 yields

\[
\min_{j=1,n} \left( e_{ij} \frac{E_i(d_{ik})}{E_i} / \lambda_j(t) + x_{ij} \frac{X_i(d_{ik})}{X_i} / \tau_j(t) \right)
\]

which is an estimation of the minimum elapsed time which we can reasonably expect to achieve at time \( t \) for instance \( k \) of task \( i \), and the \( j \) which minimizes this expression is the correct choice of host to which to send the task.

It is hoped that the successive application of this minimizing function to the tasks of program \( P \) will yield the sequence of hosts to execute those tasks which will result in the lowest elapsed time for the execution of program \( P \) which we can reasonably expect to achieve.
CHAPTER 5

Computational Model

Returning to expression 4.9 on page 23:
\[
\min_{j = 1, \ldots, m} \left( e_{ij}^{l} \frac{E_i(d_{ik})}{E_{ik}} + X_i(d_{ik}) \frac{X_{ij}}{X_i} \right) \tag{5.1}
\]

we wish to describe a computational algorithm which will compute the \( j \) which minimizes this expression. To this end, we make the following assumptions:

- \( t \) is now.
- tasks are indexed by \( i \).
- datasets are indexed by \( i \) and \( k \).
- \texttt{bench\_e\_time} is a two-dimensional array containing dedicated system execution times on the benchmark dataset for the cross product of tasks and hosts.
- \texttt{bench\_data} is a one-dimensional array of benchmark datasets corresponding to \texttt{bench\_e\_time}.
- function \texttt{complexity}(i, k) computes the complexity function of task \( i \) on dataset \( k \).
- function \texttt{load}(j) returns the current load (as a percentage of host availability) of host \( j \).
- function \texttt{sizeof}(i, k) computes the size of task \( i \)'s \( k \)'th dataset.
- function \texttt{traffic}(j) returns the current traffic (as a percentage of network availability) of the connection to host \( j \).
so that:

- $e'_{ij} = \text{bench\_e\_time}(i, j)$
- $E_i = \text{complexity}(i, \text{bench\_data}(i))$
- $E_i(d_{ik}) = \text{complexity}(i, k)$
- $\lambda_j(t) = \text{load}(j)$
- $x'_{ij} = \text{bench\_x\_time}(i, j)$
- $X_i = \text{sizeof}(i, \text{bench\_data}(i))$
- $X_i(d_{ik}) = \text{sizeof}(i, k)$
- $\tau_j(t) = \text{traffic}(j)$

This leads to the pseudocode found in figure 5-1. Function \texttt{choose\_host()} accepts a task and its data and returns the index of the host which is likely to execute the

```python
def choose_host(task, data):
    min_elapsed = +INFINITY
    min_index = -1
    for j = 1 to n
        e_time = (\text{benchmark\_e\_time}(task, j) *
            (\text{complexity}(task, data) / \text{complexity}(task, \text{benchmark\_data}(task)))) / \text{load}(j)
        x_time = (\text{benchmark\_x\_time}(data, j) *
            (\text{sizeof}(data) / \text{sizeof(\text{benchmark\_data}(task)})) / \text{traffic}(j)
        elapsed_time = e_time + x_time
        if (elapsed_time < min_elapsed) then
            min_elapsed = elapsed_time
            min_index = j
        endif
    endfor
    return(min_index)
end_function
```

\textbf{Figure 5.1: Pseudocode for choose\_host()}

This task in the lowest elapsed time. Note that the concept of ‘data’ is being very loosely
applied and is used to mean something which can provide size information for data transfer as well as size information for task complexity (see Figure 5.2.) This will be worked out in the implementation.

Function \texttt{choose\_host()} relies on several other functions. The most important of these being \texttt{complexity()}. Pseudocode for this function appears in figure 5-2.

\begin{verbatim}
function complexity(task, data)
    case task of
        1 : return (cfunc1(sizeof\_prob(data))
        2 : return (cfunc2(sizeof\_prob(data))
        .
        .
        m : return (cfuncm(sizeof\_prob(data))
    end_case
end_function
\end{verbatim}

\textbf{Figure 5.2: Pseudocode for complexity()}

Note that each task is presumed to have its own complexity function, \texttt{cfunc\#}, which operates on the \textit{size} of the current problem. These \texttt{cfunc\#s} must be individually coded for each task by a programmer that knows something about the complexity of the associated task. At a minimum, the \texttt{cfunc\#} function should return a loosely-bounded worst-case running time for \texttt{task\#’s} algorithm. At best, \texttt{cfunc\#} should provide the exact running time or an asymptotic upper-bound. For instance, the complexity function for something like the bubble sort, an $O(n^2)$ algorithm, would be simply:

\begin{verbatim}
function cfunc1(size)
    return (size^size)
end_function
\end{verbatim}

Again, although it is implied that the size of the problem, for the purpose of estimating
its complexity, can be derived from the data itself, this aspect of the algorithm will not be dealt with until the implementation.

In addition to the `complexity()` function, a `sizeof()` function is required, which is simply a byte count of the dataset provided to it as an argument. And finally, `traffic(j)`, returns the network load between the local host and host $j$. Although, with network transfer time, we have thus far proceeded with an arrangement which is symmetric to execution time, the idea of network traffic and its relationship to transfer time doesn't play quite as well as machine load. In the implementation, therefore, this notion will be modified somewhat and the estimated transfer time achieved in a more realistic way.

Selecting the best host to execute a particular task is the first half of the problem. The other half involves sending the task's arguments to the selected host, causing the task to be executed there and then returning its results to the local host. To this end, routine `choose_host()` will be used in the way illustrated by figure 5-3, where `task()`

```plaintext
function task(data)
    host = choose_host(task, data)
    results = netcall(host, task, data)
    return (results)
end_function
```

**Figure 5.3: Pseudocode for task()**

will be the local, stub version of the desired subroutine. The subroutine which actually implements the desired task will be located on some other host in the network.

In other words, and to illustrate the foregoing pseudocode, after `choose_host()` has located the best host to execute `task`, `netcall()` will
transfer (and possibly convert) its data to the selected host, cause task to be executed with that data and then return the results to the local subroutine. netcall(), and the other portions of select_host() which deal with interprocessor communication, will be implemented with an existing IPC protocol, Sun's Remote Procedure Call (RPC), the details of which will be discussed in Chapter 6, Implementation.

Although the core of this thesis, and of the implementation, is the choose_host() function, something must be said about the subroutine binding model which will be employed by the implementation. Figure 5-4 illustrates the

![Figure 5.4: Conventional Subroutine Model](image)

conventional model, which assumes a single host with a single source file where references to sub_a, sub_b and sub_c are resolved within. These routines will be statically bound. A slightly different model of subroutine binding, and one which allows for dynamic binding, involves a separately compiled library of subroutine references.
This model appears in figure 5-5.

Figure 5.5: Separately Compiled Library

The libHCS model of subroutine binding, which is also a dynamic scheme, allows for subroutine libraries to reside not only in separately compiled libraries, but on other hosts in an arbitrary network. In fact, the same library will reside on many different hosts at the same time and, depending on the loading characteristics of the distributed computer (as mentioned before) the host whose library which will execute the desired subroutine will change from one invocation to the next. A simplified illustration of this
model of subroutine binding appears in figure 5-6 This is the model which, with slight

![Diagram of libHCS Model of Distributed Computing (simplified)](image)

Figure 5.6: libHCS Model of Distributed Computing (simplified)

modifications, will be used in the implementation of libHCS/HCSdaemon.
CHAPTER 6

Implementation

As mentioned earlier, we are describing a software system which will automatically distribute tasks in a loosely-coupled, heterogeneous computer system. Task distribution will be handled by a library, libHCS, which will be linked with user programs on the local host. While task execution will be accomplished remotely by a cooperating daemon process, HCSdaemon, which will execute on all participating hosts. The choice of which tasks to distribute will be strictly dependent on the contents of libHCS, i.e. only subroutines which appear in the library will be distributed. The choice of hosts to which to send tasks will be determined by a list of candidates available to the local host\textsuperscript{1}. Binding of local calls to remote executions will take place at run time, with the process on the local host, through support routines contained in libHCS, initiating all IPC transactions. In this way, libHCS and HCSdaemon will operate in a client/server relation.

6.1 RPC

Sun Microsystem's Remote Procedure Call mechanism will provide the function-call semantics for remote (as well as local, should the local host be selected) interprocess

\textsuperscript{1} A better method would be to have a host which desires to participate broadcast on the network, but this would limit participants to those locally connected only.
communication. RPC is a message-passing scheme in which the local process sends a message, consisting of a procedure ID and procedure arguments encoded in a single argument structure, to a remote process. The remote process causes the requested procedure to be executed and then sends a message back to the local process containing the execution results encoded as a single result structure (see Figure 6.1).

![Figure 6.1: RPC Thread of Control](image)

In our case, the local, calling process will be a user program which has been linked with libHCS, and the remote process will be one of the copies of the HCSdaemon program running on all remote hosts in the distributed computer.

### 6.2 libHCS and HCSdaemon

To this end, routines in libHCS will provide the following services to a user's program:

- Resolve selected library calls
- Choose the best host to which to send tasks
- Initiate IPC with HCSdaemon on the chosen host
- Initiate transfer of arguments and procedure ID (via RPC)
• Wait for, and return, results of RPC
• Cache global network state, with periodic updates

While each instance of the HCSdaemon will provide the following:
• System load of its host
• Current benchmark data for all routines available on its host (as well as optional benchmarking of routines to produce new data)
• Network latency information (indirectly, via timed buffer transfers)
• RPC execution of requested procedures and return of results

6.3 Task Distribution

Task distribution will be transparent to the programmer. In fact, from the programmer’s point of view, there will be no additional steps other than linking with libHCS. To achieve this, programs will be coded (this implementation is done in ANSI C) and built normally; however, during the link phase, calls to routines found in libHCS will be resolved there instead of to the system’s default libraries. The routines in libHCS will be just subroutine stubs. At run time, support routines in libHCS will provide the binding to the executable version of the routine in HCSdaemon on some (possibly even the local) host in order to effect execution.

The following steps have been implemented to carry out the distribution of one of the tasks known to libHCS (see Figure 6.2 on page 34, and Figure 6.3 on page 35):

• The user’s program makes a call to sub_a.
• The procedure stub, sub_a, appears in libHCS and so is resolved there by the loader.
• Procedure stub sub_a calls choose_host() to make a choice of host.
• Stub sub_a then makes a call to the client-side RPC interface (in libHCS_clnt.c) which, in turn, initiates a network call, via the RPC clnt_call(), to the HCS-
daemon on the selected host, specifying remote procedure ID sub_a1. Data is implicitly converted to portable format and also sent to the host in the same call.

- The HCSdaemon on the remote host interprets the procedure ID for sub_a1, decodes the incoming arguments, executes the code corresponding to the actual routine sub_a and returns the results.

![Diagram](image-url)

Figure 6.2: libHCS Model of Distributed Computing
user program

```
.c call sub_a
```

libHCS.c

```
sub_a
    call choose_host()
    call sub_a_I()
```

libHCS_clnt.c

```
sub_a_l
    call clnt_call(SUB_A,
```

NETWORK

libHCS_svc.c

```
sub_a_l
    call sub_a
```

libHCS_l.c

```
sub a (code)
    return
```

Figure 6.3: libHCS Thread of Control
The local sub_a receives and decodes the results from the HCSdaemon on the remote host and returns them to the user.

Recall that the user's program will have calls to distributable subroutines that look the same as if the call was to the normal library version of the subroutine, i.e.:

\[ \text{result} = \text{sub}_a(\text{arg}_1, \text{arg}_2, \text{arg}_3, \ldots, \text{arg}_n); \]

These calls will be resolved by the link editor to the stub entries in libHCS (see Figure 6.4 for the framework of one of these stubs.):

```c
sub_a_res sub_a(arg_1, arg_2, ..., arg_n)
    type_1 arg_1;
    type_2 arg_2;
    ...
    type_n arg_n;
{
    CLIENT *clnt_handlep;
    int choose_host();
    sub_a_arg arg;
    sub_a_res res;
    /*
    * Insert code to copy sub_a arguments and
    * global variables to its arg structure
    */
    ...
    if (choose_host(SUB_A, prob_size, sizeof(arg),
                    sizeof(res), &clnt_handlep) == 1) {
        fprintf(stderr,"sub_a: choose_host failed\n");
        exit(1);
    }

    res = *sub_a_1(&arg, clnt_handlep);
    /*
    * Insert code to copy sub_a results and
    * global variables from its res structure
    */
    ...
    return;
}
```

Figure 6.4: Typical libHCS Subroutine Stub
choose_host()

Note that the stub calls function choose_host(), a key element of the libHCS system. choose_host() is called by each subroutine, sub(), in the library to discover the best host to execute the current instance of sub(). The following steps are performed on behalf of the subroutine by choose_host():

1. Discover which hosts will be participating (done the first time through.)
2. Establish RPC communication with each host.
3. Poll each available host to discover its load as well as network latency to the host.
4. Acquire from each available host the benchmark data for each routine.
5. Calculate the score for each routine called (the sum of the estimated time to transfer data, execute the routine on the current input, and return results) for each host.
6. Pick the host with the lowest score and return its RPC handle.

procedure choose_host(sub,arg)
  if (first_pass)
    get_hostlist()
    for host in hostlist
      establish_communication(host)
    end_for
  end_if
  if (POLL_TIME)
    load[host] = get_load(host)
    network_latency[host] = get_network_latency(host)
    transfer_latency[host] = get_transfer_latency(host)
    bench_data[host,sub] = get_benchmark_data(host,sub)
  end_if
  for host in hostlist
    exec_time[host,sub] =
      estimate_exec(host,sub,arg,bench_data(host,sub))
    xfer_time[host,arg] = estimate_xfer(host, arg)
    score[sub,host] =
      exec_time[host,sub] + xfer_time[host, arg]
  end_for
  min_host = find_min_host(score)
  return (min_host)
end_procedure

Figure 6.5: Pseudocode for choose_host()
Pseudocode for the implementation appears in Figure 6.5 on page 37, however, since choose_host() is fairly long, its entire implementation will not be reproduced here. Please refer to Appendix B, libHCS, for full details.

6.5 rpcgen(l)

As mentioned before, the IPC portion of this implementation will be accomplished using the ONC RPC Library. Construction of a distributed application with this library is facilitated by the rpcgen(1) protocol compiler provided with the library. Rpcgen accepts as input a protocol definition file - in our case libHCS.x (see Appendix A, libHCS.x). Compiling this file produces four new source files:

- **libHCS.h** A header file with: program constants, XDR structure definitions and typedefs, RPC Program Number, Version Number and Procedure IDs.
- **libHCS_clnt.c** The client-side interface containing calls to the RPC `clnt_call()` routine to cause remote procedure execution.
- **libHCS_svc.c** The source file containing the main program for the server-side of our RPC client/server pair. This program receives RPC requests and invokes the relevant procedure.
- **libHCS_xdr.c** XDR file containing the eXternal Data Representation routines for inter-architecture data transfer of the data types defined in libHCS.h.

which contain the program framework and necessary calls to implement an RPC client/server process pair. An illustration of this process appears in Figure 6.6 on page 39.

Since the RPC protocol is free of transport dependencies, the choice of underlying transport can be anything. TCP, a reliable transport which allows for messages of arbitrary length, has been chosen as the transport. This choice simplifies the prototype implementation considerably.
In addition to the foregoing source files are added:

- `libHCS.c` The bridge between user calls and the client-side RPC interface found in `libHCS_clnt.c`.
- `libHCS_1.c` The source file where the actual subroutine code is located.

### 6.6 Building the System

`libHCS.h`, `libHCS.c`, `libHCS_clnt.c` and `libHCS_xdr.c` are compiled to yield `libHCS.a`; while `libHCS.h`, `libHCS_svc.c`, `libHCS_1.c` and `libHCS_xdr.c` are compiled to produce `HCSdaemon`. These operations are illustrated in Figure 6.7. (The complete source codes for `libHCS.x`, `libHCS.a` and `HCSdaemon` appear in the Appendixes.)
Building the `libHCS.a` library
(to be linked with an RPC client)

Building the HCSdaemon program
(an RPC Server)

Figure 6.7: Building libHCS and HCSdaemon
To create a distributed application, the user builds his or her own program normally, linking libHCS to it (see Figure 6.8).

![Figure 6.8: Linking a User’s Program](image)

6.7 Execution

To execute the resulting distributed system, HCSdaemons are started on each participating host. Each daemon registers itself with its local portmapper (so that RPC requests can find it) and then goes into a wait loop, listening for requests. When the user starts his or her program, the RPC traffic begins to flow between userprog (libHCS) and each of the HCSdaemons (Note: In the prototype implementation, a file of participating hosts, called hostlist, is required to exist in the same directory as the user's executable program.) For additional details on RPC, please refer to the complete RPC
documentation. Also, please see Figure 6.2 on page 34.

6.8 Example Routine

As a small example of the individual elements which comprise a distributed routine, we will examine one of the three routines implemented in the prototype version of libHCS/HCSdaemon (see Chapter 7 on page 51 for additional details.) First of all, let's take a look at the user program which calls these routines (Figure 6.9):

```c
#include <stdio.h>

float addf(float x, float y);
float mulf(float x, float y);
float divf(float x, float y);

main(
{
    float x,y,z;
    x = 9.0;
    y = -2.3;
    z = addf(x,y);
    printf("x = %f, y = %f, addf(x,y) = %f\n", x,y,z);
    z = mulf(x,y);
    printf("x = %f, y = %f, mulf(x,y) = %f\n", x,y,z);
    z = divf(x,y);
    printf("x = %f, y = %f, divf(x,y) = %f\n", x,y,z);
    z = (addf(x,y) - mulf(x,y)) * divf(x,y);
    printf("(addf(x,y) - mulf(x,y)) * divf(x,y) = %f\n",z);
    z = (addf(mulf(x,y),divf(x,y)) * \n        divf(addf(x,y)+mulf(x,y),y)) + divf(addf(x,y)+mulf(x,y),y) = %f\n",z);
}
```

**Figure 6.9: Sample User Program**

where `addf()`, `mulf()` and `divf()` perform the operations suggested by their names.
Corresponding to the \texttt{addf()} call, the following stub appears in \texttt{libHCS.c}. This will be the loader's resolution of the user's \texttt{addf()} call (see Figure 6.10).

```c
float addf(float x, float y)
{
    CLIENT *clnt_handlep;
    int choose_host();
    addf_arg arg;
    addf_res res;

    arg.x = x;
    arg.y = y;

    if (choose_host(ADDF,1,sizeof(arg),
                   sizeof(res),&clnt_handlep) == 1) {
        fprintf(stderr,"addf: choose_host failed\n");
        exit(1);
    }
    res = *addf_l(&arg,clnt_handlep);
    return (res.z);
}
```

\textbf{Figure 6.10: Subroutine \texttt{addf()} in \texttt{libHCS.c}}

Next comes the client-side interface to \texttt{addf()}. \texttt{addf_l()} appears in

```c
addf_res *
addf_l(argp, clnt)
addf_arg *argp;
CLIENT *clnt;
{
    static addf_res res;
    (void) memset((char *)&res, 0, sizeof (res));

    if (clnt_call(clnt, ADDF, xdr_addf_arg, argp,
                  xdr_addf_res, &res, OK_TIMEOUT) != RPC_SUCCESS) {
        clnt_perror(clnt,"addf_l");
        return (NULL);
    }
    return (&res);
}
```

\textbf{Figure 6.11: \texttt{addf_l()} from \texttt{libHCS_clnt.c}}
libHCS_clnt.c.

Of interest in \texttt{addf} () are the custom XDR filter specifications, \texttt{xdr\_addf\_arg()} and \texttt{xdr\_addf\_res()} (see Figure 6.12):

\begin{verbatim}
bool_t
xdr_addf_arg(xdrs, objp)
    XDR *xdrs;
    addf_arg *objp;
{
    if (!xdr_float(xdrs, &objp->x)) {
        return (FALSE);
    }
    if (!xdr_float(xdrs, &objp->y)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_addf_res(xdrs, objp)
    XDR *xdrs;
    addf_res *objp;
{
    if (!xdr_float(xdrs, &objp->z)) {
        return (FALSE);
    }
    return (TRUE);
}
\end{verbatim}

\textbf{Figure 6.12: addf ()'s XDR Routines}

which effect the host-independent conversion and transfer of the arguments and results of the \texttt{addf} () routine. By using the XDR filters for the XDR Standard Primitive and Composite types, one can build filters for data types of arbitrary complexity. This is a simple example. For a more interesting example see the libHCS\_xdr.c source in Appendix B.

In the HCSdaemon file, libHCS\_svc.c has a mechanism for the uniform treatment
of all procedure requests which is based on the unique procedure ID. In Figure 6.13, note
that ADDF is a globally available #define which specifies the procedure ID. This is
one of the values passed in the call to clnt_call() (see Figure 6.11). In the
following, only the entries in libHCS_svc.c relevant to addf() appear (see
Figure 6.13). Please see the libHCS_svc.c file in Appendix C for the full context.

```c
static void
hcsdaemon_l(rqstp, transp)
    struct svc_req *rqstp;
    SVCXPRT *transp;
{
    union {
        ...
        addf_arg addf_l_arg;
        ...
    } argument;
    char *result;
    bool_t (*xdr_argument)(), (*xdr_result)();
    char *(*local)();

    switch (rqstp->rq_proc) {
    ...
    case ADDF:
        xdr_argument = xdr_addf_arg;
        xdr_result = xdr_float;
        local = (char (*)(*)) addf_l;
        break;
    ...
    default:
        svcerr_noproc(transp);
        return;
    }

Figure 6.13: addf() 's Relevant Entries in libHCS_svc.c
```

Finally, the actual executable code for addf() appears in libHCS_l.c (see
Figure 6.14: \texttt{addf\_1()} in \texttt{libHCS\_1.c}

6.9 Extensibility

Of particular interest in the implementation is extensibility. Routines can be added to \texttt{libHCS/HCSdaemon} by following the steps outlined below (also, please refer back to the most recent series of figures). To add a new routine to \texttt{libHCS/HCSdaemon}, additions or changes must be made to all source files in \texttt{libHCS/HCSdaemon}:

- \texttt{libHCS.x}
- \texttt{libHCS.h}
- \texttt{libHCS.c}
- \texttt{libHCS\_clnt.c}
- \texttt{libHCS\_svc.c}
- \texttt{libHCS\_xdr.c}
- \texttt{libHCS\_1.c}

To add a new entry, \texttt{func()}, start by making an addition to \texttt{libHCS.x}:

1. Collect \texttt{func()}'s argument list into a single \texttt{c} structure. Include any global variables that must be passed. Name the structure - \texttt{func\_arg}.
2. Collect all variables used to return values to the calling program into another \texttt{c}
structure. Duplicate any pass-by-reference items appearing in the argument list, as well as any global variables, and the function’s original return value. Name the structure - func_res.

3. Add a new procedure specification: func_res FUNC(func_arg) = #; where ‘#’ is the next procedure number in the list.

Following these additions to libHCS.x, and in a safe place away from the production libHCS/HCSdaemon code, compile the modified libHCS.x file with rpcgen.

Then make changes to the rest of the source files as follows:

4. Use libHCS.h as it comes from rpcgen.

5. Add the func() function definition to libHCS.c as follows:

   a. Duplicate the original function’s prototype
   b. Add the declarations for function ‘func’

      CLIENT *clnt_handlep;
      int choose_host();
      func_arg arg;
      func_res res;

   c. Copy arguments to the routine’s ‘arg’ data structure
   d. Add a call to ‘choose_host’

      if (choose_host(FUNC,prob_size,arg_size,
                      res_size,&clnt__handlep) == 1) {
          printf(stderr,"func:choose_host failed\n");
          exit(1);
      }

   The arguments provided to this call assume the programmer has some basic knowledge about the current invocation -

      prob_size = current ‘size’ of the problem (for estimation of its complexity.
      arg_size = size of argument data to be sent.
      res_size = size of results data to be returned.

   e. Call the network version of the routine with the returned client handle

      res = *func_l(&arg,clnt_handlep);
f. Copy, if necessary, any return values to local variables.

g. Free, if necessary, any result arrays implicitly allocated by the XDR routines.

h. Return, if necessary, the value of the function.

6. From the libHCS_clnt.c file just produced extract the func_l() definition and include it in the production libHCS_clnt.c file.

7. From the libHCS_svc.c file just produced extract the following from hcsdaemon_l() and include at the corresponding locations in this file:

   in the union 'argument' include the new entry:

   func_arg func_l_arg;

   in the switch (reqstp->rq_proc) include the new entry

   case FUNC:
   
   xdr_argument = xdr_func_arg;
   xdr_result = xdr_func_res;
   local = (char *(*)()) func_l;
   break;

8. Use libHCS_xdr.c as it comes from rpcgen.

9. Make an addition to the libHCS_l.c file. The following is reproduced from that file:

/*
 * The following commentary describe the basic steps
 * for converting an existing function to this system.
 *
 * The new function declaration is uniform and simple.
 * The argument list and return type have been replaced
 * by their RPC versions (which look, essentially, the
 * same for all routines.)

func_res *func_l(struct func_arg *arg)
{

 * Within the body of the function, the original arguments
 * are reproduced as automatic variables. Also, a results
 * structure is allocated.

 float a,b,c;
 static func_res res;
Then, the next step is to assign to the local variables the corresponding fields from the argument structure so that the balance of the code will behave as if these values had been passed in normally via the argument list.

```c
a = arg->a;
b = arg->b;
c = arg->c;
```

A timing call is made in case we are benchmarking.

```c
times(&before);
```

The body of the original subroutine appears next.

```c
...
```

Followed by the balance of the benchmarking addition

```c
times(&after);
```

```c
if (local_benchmark) benchmark_1(FUNC,&before,&after,1);
```

Finally, at a common return point (see some of the more complicated examples below for a better example) the return values are assigned to the relevant fields in the results structure before it is returned.

```c
res.d = d;
return(&res);
```

And that's it.

10. Rebuild the system.

6.10 Benchmarking

A topic of interest in the implementation is the benchmarking mechanism which has been built into HCSdaemon. Recall that HCSdaemon must provide prospective clients with benchmark data for all routines it knows about. This benchmark information will be produced automatically, if it doesn't already exist, or by request with the -b
option on the HCSdaemon invocation. More detail on this aspect of the implementation will be provided in Chapter 7, Implementation Case Study.

6.11 Final Issues

It must be noted that the following issues have been ignored in the prototype implementation:

- Network security.
- Network or host failures.
- General error handling.
- Differences in numerical accuracy.
CHAPTER 7

Implementation Case Study

As previously alluded to, the implementation of this system proceeded in several phases. Those phases were:

1. A choice was made of interprocess communication mechanism.
2. A framework was established for making a Remote Procedure Call to a dynamically selected host.
3. choose_host() was written to make the best choice of host.
4. A small set of 'easy' functions was coded into a prototype system.
5. Automated benchmarking was added.
6. An existing, 'real world' set of subroutines was ported to this system.

In Chapter 6 we explored phases 1-3. In this chapter we will examine phases 4-6.

7.1 Prototype

After the initial work of selecting an IPC protocol and designing and programming a generalized mechanism for dynamically selecting and calling a remote host to execute an RPC, a set of 3 functions was ported to the resulting system. A single source file, containing the source code for these three functions as well as a main program which makes calls to them appears in Figure 7.1 on page 52. Output from
execution of this program appears in Figure 7.2 on page 53. The process of creating this

```c
#include <stdio.h>
float addf(float x, float y);
float mulf(float x, float y);
float divf(float x, float y);

main()
{
    float x, y, z;
    x = 9.0;
    y = -2.3;
    z = addf(x, y);
    printf("x = %f, y = %f, addf(x,y) = %f\n", x, y, z);
    z = mulf(x, y);
    printf("x = %f, y = %f, mulf(x,y) = %f\n", x, y, z);
    z = divf(x, y);
    printf("x = %f, y = %f, divf(x,y) = %f\n", x, y, z);
    z = (addf(x, y) - mulf(x, y)) * divf(x, y);
    printf("(addf(x,y) - mulf(x,y)) * divf(x,y) = %f\n", z);
    z = (addf(mulf(x, y), divf(x, y)) * \
    divf(addf(x, y) + mulf(x, y), y));
    printf("(addf(mulf(x,y),divf(x,y)) * \ 
    divf(addf(x,y) + mulf(x,y),y)) = %f\n", z);
}

float addf(float x, float y)
{
    return(x + y);
}
float mulf(float x, float y)
{
    return(x * y);
}
float divf(float x, float y)
{
    return(x / y);
}
```

Figure 7.1: myprog.c
prototype was simply the first application of the extensibility process described in

\[
x = 9.000000, y = -2.300000, \text{addf}(x,y) = 6.700000 \\
x = 9.000000, y = -2.300000, \text{mulf}(x,y) = -20.699999 \\
x = 9.000000, y = -2.300000, \text{divf}(x,y) = -3.913043 \\
(\text{addf}(x,y) - \text{mulf}(x,y)) \cdot \text{divf}(x,y) = -107.217384 \\
(\text{addf}(\text{mulf}(x,y), \text{divf}(x,y))) \cdot \\
\text{divf}(\text{addf}(x,y) + \text{mulf}(x,y), y) = -149.818512
\]

Figure 7.2: Output from myprog.c

Section 6.9 on page 46. This process was applied to all three of these functions and the resulting system executed across 4 hosts (see Table 7.1). In the prototype there was no automated benchmarking of routines, so artificial benchmark numbers were provided to cause some variability in host selection (otherwise, the local host would have always been selected, due to the simple nature of the task.) Performance results of the

<table>
<thead>
<tr>
<th>host</th>
<th>architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>palantir</td>
<td>sun sparcstation</td>
</tr>
<tr>
<td>nye</td>
<td>sun 4/690 sparcserver</td>
</tr>
<tr>
<td>aurora</td>
<td>Convex C-220</td>
</tr>
<tr>
<td>clark</td>
<td>Cray Y-MP2/216</td>
</tr>
</tbody>
</table>

Table 7.1: Hosts in the Distributed Computer

distributed program would be meaningless and will not be evaluated.

7.2 Automated Benchmarking

One of the most important features of this system must be the ease with which performance data may be acquired for any given host-task pair. Certainly, HCSdaemon should be to able detect if no performance data exists and correct the situation. Also, it should be possible to instruct HCSdaemon to replace existing performance data with
more recent data in case conditions have changed sufficiently, or existing performance
data is insufficiently representative.

This benchmarking process is achieved within each routine in libHCS_1.c (see
Appendix C on page 111) with: a flag (local_benchmark), indicating that
benchmark timing should be performed; timing calls that bracket the entry and exit
points of the computation portion of the routine; and a call to benchmark_l() to
record the data (see Figure 7.3.) Note that the parameter size must be supplied by the

/* A timing call is made in case we are benchmarking. */
times(&before);

/* The body of the original subroutine appears next. */
...

/* Followed by the balance of the benchmarking code. */
if (local_benchmark) {
    times(&after);
    benchmark_l(FUNC,&before,&after,size);
}
...

Figure 7.3: Benchmarking Code

programmer of the routine based on some value present in the current execution (e.g. an
array size or loop count.). The benchmark data is maintained in a separate file, libHCS.b,
consisting of one line per routine. Where each line contains 4 fields:

• Task ID
• Task Size
• Complexity (see libHCS.h for the mapping)
• Task Execution Time
This file is read by HCSdaemon at start-up (see Figure 7.4 for a sample libHCS.b).

```
0  1  1  -1.000000
1  1  1  0.000020
2  1  1  0.000019
3  1  1  0.000020
4  1  2  -1.000000
5  100 4  0.000877
6  100 4  0.022376
```

Figure 7.4: Sample libHCS.b

### 7.3 clinpack

Following the successful prototype and benchmark phases, a more realistic set of subroutines was needed for the next, and final step. The *clinpack* benchmark was selected because it is well-known, manageably sized and portable. *clinpack* is a C language program derived from LINPACK [22], written by Jack Dongarra in March of 1978. It was ported to the C language by Bonnie Toy in May of 1988 (as unpublished source code). The clinpack source is available via anonymous ftp from a number of ftp

<table>
<thead>
<tr>
<th>clinpack routine</th>
<th>what it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>daxpy</code></td>
<td>constant times a vector plus a vector</td>
</tr>
<tr>
<td><code>ddot</code></td>
<td>forms the dot product of two vectors</td>
</tr>
<tr>
<td><code>dgefa</code></td>
<td>factors a double precision matrix by gaussian elimination</td>
</tr>
<tr>
<td><code>dgesl</code></td>
<td>solves the double precision system <code>a * x = b</code> or <code>trans(a) * x = b</code> using the factors computed by <code>dgeco</code> or <code>dgefa</code></td>
</tr>
<tr>
<td><code>dmxpy</code></td>
<td>multiply matrix <code>m</code> times vector <code>x</code> and add the result to vector <code>y</code></td>
</tr>
<tr>
<td><code>dscal</code></td>
<td>scales a vector by a constant</td>
</tr>
</tbody>
</table>

Table 7.2: Candidate Routines in *clinpack*
sites and will not be reproduced here, or in the Appendixes (except for the portions which have been converted to libHCS/HCSdaemon.) Candidate subroutines from *clinpack*, along with their descriptions, appear in Table 7.2.

After examination of the source code, the *daxpy* routine was, naively, first elected for conversion because of its apparent simplicity, primarily in terms of parameter passing. However, after a successful port, it was discovered that this routine is called approximately 180,000 times over the course of the benchmark, and that its execution time is extremely low compared to the time to transfer data\(^1\). These facts made it unsuitable for use in the system and its functionality was returned to the local library.

Next, the two routines *dgefa* and *dgesl* were ported. These proved to be much more suitable for this purpose. The data transfer requirements were the same or less while the execution time was considerably greater than *daxpy*.

The resulting distributed system was executed on the 4 hosts in Table 7.1. A fifth host was later added:

<table>
<thead>
<tr>
<th>host</th>
<th>architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>saturn</td>
<td>Silicon Graphics</td>
</tr>
</tbody>
</table>

Table 7.3: Additional Host in the Distributed Computer

\(^1\) A simple reading of the LINPACK Users’ Guide would have saved this step. It is clear from that document that the best candidates are *dgefa* and *dgesl*.  

<table>
<thead>
<tr>
<th>clinpack routine</th>
<th>what it does</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>idamax</em></td>
<td>finds the index of element having max. absolute value</td>
</tr>
<tr>
<td><em>matgen</em></td>
<td>generate a random matrix</td>
</tr>
</tbody>
</table>

Table 7.2: Candidate Routines in *clinpack*
As can be seen from the resulting performance numbers (see Table 7.5 and Table 7.6) the performance of the resulting distributed system, although slightly faster for a couple of the 200 element arrays, did not improve on that of the native benchmark.

```
palantir % ./clinpackH
Rolled Single Precision Linpack
  norm. resid resid machep x[0]-1 x[n-1]-1
  1.2    2.9e-05 1.2e-07 -2.7e-05 -1.7e-05
  times are reported for matrices of order 100
  dgefa dgesl total kflops unit ratio
times for array with leading dimension of 201
  0.77 0.05 0.82 837 2.39 14.64
  0.31 0.07 0.38 1807 1.11 6.79
  0.31 0.07 0.38 1807 1.11 6.79
  0.29 0.06 0.36 1918 1.04 6.39
  times for array with leading dimension of 200
  0.28 0.07 0.35 1962 1.02 6.25
  0.31 0.07 0.38 1807 1.11 6.79
  0.26 0.06 0.32 2146 0.93 5.71
  0.30 0.07 0.37 1881 1.06 6.52
  Rolled Single Precision 1881 Kflops ; 10 Reps
```

**Figure 7.5: Execution of Distributed clinpack from palantir**

```
palantir % ./clinpack
Rolled Single Precision Linpack
  norm. resid resid machep x[0]-1 x[n-1]-1
  1.6    3.8e-05 1.1e-07 -1.3e-05 -7.5e-06
  times are reported for matrices of order 100
dgefa dgesl total kflops unit ratio
times for array with leading dimension of 201
  0.35 0.01 0.36 1907 1.05 6.43
  0.34 0.01 0.35 1962 1.02 6.25
  0.35 0.01 0.36 1907 1.05 6.43
  0.35 0.01 0.36 1923 1.04 6.37
  times for array with leading dimension of 200
  0.35 0.01 0.36 1907 1.05 6.43
  0.35 0.01 0.36 1907 1.05 6.43
  0.34 0.01 0.35 1962 1.02 6.25
  0.35 0.01 0.36 1923 1.04 6.37
  Rolled Single Precision 1923 Kflops ; 10 Reps
```

**Figure 7.6: Execution of Native clinpack on palantir**
This is neither surprising nor disappointing, since there has been no work on optimization of the distributed portions of the code whatsoever. Also it must be pointed out that, at this stage of development, feasibility is the key aspect to be demonstrated, not performance. Additional work in this area is indicated.
CHAPTER 8

Simulation

To assess the validity of the assumption made in Chapter 4, Abstract Model, that good scheduling choices can be made even in the presence of changing loads, the portion of libHCS/HCdaemon relevant to host scheduling \( \text{choose\_host()} \) was extracted from the system source code and placed into a new program, \texttt{simu.c} (Appendix D on page 136.) This program seeks to reproduce libHCS scheduling behavior by randomly simulating the external factors (host and network loads) affecting execution of a simple program by libHCS/HCdaemon. Also, code was included to allow simulations to be performed under the same load conditions with the following additional host choice policies: selection at random; selection of strongest host (i.e. the host with the best benchmark for the task); and selection of least loaded host.

8.1 Pseudocode

Pseudocode representing the procedure used by the simulator program appears in Figure 8.1 on page 60, where you will see that the simulator is executed with new random number sequences for each of a large number of trials (MAXTRIALS). Notice that random number 'streams' were employed so that for each trial a random sequence of load changes may be repeated for execution of the program with each of the MAXCONFIG \( (2^\text{MAXHOSTS} - 1) \) possible configurations of the MAXHOSTS hosts.
Pseudocode for the modified `choose_host()` function appears in Figure 8.4 on page 64, while pseudocode for the 'subroutine' appears in Figure 8.3. In reference to the program simulator:

```c
for k = 1, MAXTRIALS
  /*
   * create and save random streams
   * (load and latency for each host)
   */
  ...
  for config = l, MAXCONFIGS
    /*
     * reinitialize random streams for each config.
     */
    ...
  for hst = 1, MAXHOSTS
    /* initialize load data */
    ...
  end_for

  /* initialize the current execution parameters */
  GlobalTime = 0.0
  first_pass = 0
  watchdog_timer = 0.0
  last_hostupdate_time = 0.0

  /* execute simulated program */
  for i = 1, 20
    call subroutine(a,...
    call subroutine(b,...
    call subroutine(c,...
    call subroutine(d,...
  end_for

  /*
   * sum the elapsed time (by configuration)
   */
  ...
  end_for
end_for
/* print results */
...
end_program
```

Figure 8.1: Pseudocode for simu.c
simulate exec_time:

```c
exec_time = estimate_exec(sub_idx,
&hosts[host_idx], size);
while (exec_time > (float)POLL_TIME) {
    update_sim((float)POLL_TIME);
    reexec_time = exec_time - (float)POLL_TIME;
    size = inverse_order(host_idx, sub_idx,
        reexec_time, exec_time)*size;
    exec_time = estimate_exec(sub_idx,
        &hosts[host_idx], size);
}
if (exec_time > 0.0)
    update_sim(exec_time);
```

Figure 8.2: exec_time Simulation Code

'simulate exec_time' statement which appears in the subroutine pseudocode, we must account for the passage of time while the subroutine executes. Our ability to accomplish this is complicated by the fact that as the load on the remote host changes, so does the execution time of the remaining portion of the algorithm. Therefore, at each passage of POLL_TIME time we need to re-estimate the remaining execution time. The source code which accomplishes this little feat appears in Figure 8.2. Notice that since the original estimation operation requires the application of the time complexity function to the current problem size, the inverse operation requires the application of the inverse of the respective time complexity function. For complete details on this aspect of the simulation, see the function inverse_order()\(^1\) in the simulation source code, page 153.

8.2 Simulation Parameters

The fixed parameters of the simulation consist of:

\(^1\) Of particular interest is the inverse of the complexity function \(n \log n\). See the source code for further details.
• MAXTRIALS = 10000
• MAXHOSTS = 4 (and thus MAXCONFIGS = 15)
• Four simulated subroutines - The cross product of subroutine to host performance
  appears in Table 8.1 (also, see Figure 8.5 on page 65), while the benchmark size

<table>
<thead>
<tr>
<th>Benchmark exec_time Matrix</th>
<th>Host a</th>
<th>Host b</th>
<th>Host c</th>
<th>Host d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subroutine 0</td>
<td>.1 sec</td>
<td>1.0 sec</td>
<td>.4 sec</td>
<td>2.0 sec</td>
</tr>
<tr>
<td>Subroutine 1</td>
<td>2.5 sec</td>
<td>10.0 sec</td>
<td>10.0 sec</td>
<td>50.0 sec</td>
</tr>
<tr>
<td>Subroutine 2</td>
<td>5.0 sec</td>
<td>50.0 sec</td>
<td>15.0 sec</td>
<td>100.0 sec</td>
</tr>
<tr>
<td>Subroutine 3</td>
<td>20.0 sec</td>
<td>200.0 sec</td>
<td>80.0 sec</td>
<td>400.0 sec</td>
</tr>
</tbody>
</table>

Table 8.1: Benchmark Execution Matrix

and complexity for each of the subroutines appears in Table 8.2.

<table>
<thead>
<tr>
<th>Benchmark Size (n)</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subroutine 0</td>
<td>n</td>
</tr>
<tr>
<td>Subroutine 1</td>
<td>nlogn</td>
</tr>
<tr>
<td>Subroutine 2</td>
<td>n²</td>
</tr>
<tr>
<td>Subroutine 3</td>
<td>n³</td>
</tr>
</tbody>
</table>

Table 8.2: Subroutine Complexity and Benchmark Size

• A ‘program’ consisting of a loop executing each of the 4 subroutines 20 times
  (see Figure 8.1.)

The dynamic aspects of the simulation consist of the randomized load on each
host as well as the randomized network and transfer latencies. These quantities were
initialized for each trial to a random value selected from a uniformly distributed space of
pseudorandom numbers and then allowed to experience quantum changes (the delta
value) over the course of the trial (see Table 8.3). The transfer latency values were

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>max</th>
<th>delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>host load (% available)</td>
<td>1</td>
<td>100</td>
<td>5</td>
</tr>
<tr>
<td>network latency (seconds)</td>
<td>.008</td>
<td>.016</td>
<td>.0001</td>
</tr>
<tr>
<td>transfer latency (seconds)</td>
<td>.008</td>
<td>.016</td>
<td>.0002</td>
</tr>
</tbody>
</table>

Table 8.3: Dynamic Parameters of the Simulation

chosen to approximate Ethernet performance on an 8K data transfer.

8.3 Statistics

Statistics were produced from each of those executions. The sorted average
equency times, by configuration, of the suite of 'programs' for the choose_host host
choice policy are displayed in the graph associated with Figure 8.4 on page 67. The other
host choice policies (each individually ordered by configuration) are similarly displayed
in Table 8.5 on page 68 through Table 8.7 on page 70. Finally, the choose_host policy is
displayed alongside the other host choice policies (ordered according to the choose_host
sort order) in the graph associated with Figure 8.4 on page 67. It should be pointed out
that the results of the choose_host policy simulation show very large standard deviations
within each configuration\(^2\); however, the trend suggested by the graph of the average

procedure subroutine(sub_idx,...
    choose_host(sub_idx,...
    simulate send_time
    simulate exec_time
    simulate recv_time
end_procedure

Figure 8.3: Pseudocode for subroutine()

\(^2\) The only policy for which this sort of statistics was computed.
execution times, is consistent across all trials. Also, there is strong correlation of the
ordering of individual trials to the ordering of the average of all trials, with 40%
displaying the identical ordering and another 40% showing the same ordering with the
exception of a single spike. In all trials, the lowest elapsed time was achieved with the
configuration containing all available hosts.

procedure choose_host(routine, size, ... 
  if (first_pass) 
    for i = 1, hostcnt 
      /* invalidate host to force a host update */ 
      ... 
    end_for 
  end_if 
  if (watchdog_timer is expired) 
    /* invalidate all hosts to force a host update */ 
    ... 
  end_if 
  for i = 1, hostcnt 
    if (host(i) is invalid but in current configuration) 
      /* update host(i) and invalidate its scores */ 
      ... 
    end_if 
  end_for 
  for i = 1, hostcnt 
    /* calculate score of this routine for host(i) */ 
    ... 
  end_for 
  /* select the best host for this routine based on 
  * score (also, for comparative purposes, selection 
  * will be made: 
  * a) at random, 
  * b) by strongest host (irrespective of score) 
  * c) by least loaded (irrespective of score) 
  */ 
end_procedure

Figure 8.4: Pseudocode for choose_host()
8.4 Caveats

The simulation is, to some extent, self-fulfilling. The calculation used to estimate the execution time is the same as the calculation used to simulate execution time. This means that our estimation algorithm is \textit{perfect} and that the results are only dependent on the load characteristics of the distributed computer. In a sense, this is exactly what we want; however, it is a departure from reality which must be factored in to our conclusions drawn from the results of the simulation.

In the simulation, all issues of start-up overhead and POLL\_TIME interval overhead have been ignored. These are immaterial for the host selection anyway, but will
figure prominently when it is time to assess the feasibility of using this system in a production environment. As for host selection overhead, it is comprised of a few simple calculations which should not affect the validity of the simulation results.
<table>
<thead>
<tr>
<th>Key</th>
<th>Configuration</th>
<th>Total seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0,0,0,1)</td>
<td>14383</td>
</tr>
<tr>
<td>2</td>
<td>(0,1,0,0)</td>
<td>7695</td>
</tr>
<tr>
<td>3</td>
<td>(0,1,0,1)</td>
<td>5733</td>
</tr>
<tr>
<td>4</td>
<td>(0,0,1,0)</td>
<td>3647</td>
</tr>
<tr>
<td>5</td>
<td>(0,0,1,1)</td>
<td>3187</td>
</tr>
<tr>
<td>6</td>
<td>(0,1,1,0)</td>
<td>3025</td>
</tr>
<tr>
<td>7</td>
<td>(0,1,1,1)</td>
<td>2887</td>
</tr>
<tr>
<td>8</td>
<td>(1,0,0,0)</td>
<td>1135</td>
</tr>
<tr>
<td>9</td>
<td>(1,0,0,1)</td>
<td>1094</td>
</tr>
<tr>
<td>10</td>
<td>(1,1,0,0)</td>
<td>990</td>
</tr>
<tr>
<td>11</td>
<td>(1,1,0,1)</td>
<td>986</td>
</tr>
<tr>
<td>12</td>
<td>(1,0,1,0)</td>
<td>821</td>
</tr>
<tr>
<td>13</td>
<td>(1,0,1,1)</td>
<td>821</td>
</tr>
<tr>
<td>14</td>
<td>(1,1,1,0)</td>
<td>815</td>
</tr>
<tr>
<td>15</td>
<td>(1,1,1,1)</td>
<td>815</td>
</tr>
</tbody>
</table>

Table 8.4: choose_host() Policy Elapsed Times
<table>
<thead>
<tr>
<th>Key</th>
<th>Configuration</th>
<th>Total seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0,0,0,1)</td>
<td>14383</td>
</tr>
<tr>
<td>2</td>
<td>(0,1,0,1)</td>
<td>12709</td>
</tr>
<tr>
<td>3</td>
<td>(0,1,1,1)</td>
<td>10438</td>
</tr>
<tr>
<td>4</td>
<td>(0,0,1,1)</td>
<td>10132</td>
</tr>
<tr>
<td>5</td>
<td>(1,1,0,1)</td>
<td>9349</td>
</tr>
<tr>
<td>6</td>
<td>(1,0,0,1)</td>
<td>8593</td>
</tr>
<tr>
<td>7</td>
<td>(1,1,1,1)</td>
<td>8498</td>
</tr>
<tr>
<td>8</td>
<td>(1,0,1,0)</td>
<td>7695</td>
</tr>
<tr>
<td>9</td>
<td>(1,0,1,1)</td>
<td>7586</td>
</tr>
<tr>
<td>10</td>
<td>(0,1,1,0)</td>
<td>6389</td>
</tr>
<tr>
<td>11</td>
<td>(1,1,1,0)</td>
<td>4971</td>
</tr>
<tr>
<td>12</td>
<td>(1,1,0,1)</td>
<td>4942</td>
</tr>
<tr>
<td>13</td>
<td>(0,0,1,0)</td>
<td>3647</td>
</tr>
<tr>
<td>14</td>
<td>(1,0,1,0)</td>
<td>2654</td>
</tr>
<tr>
<td>15</td>
<td>(1,0,0,0)</td>
<td>1135</td>
</tr>
</tbody>
</table>

Table 8.5: Random Policy Elapsed Times
Table 8.6: Least-loaded Policy Elapsed Times

<table>
<thead>
<tr>
<th>Key</th>
<th>Configuration</th>
<th>Total seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0,0,0,1)</td>
<td>14383</td>
</tr>
<tr>
<td>2</td>
<td>(0,1,0,0)</td>
<td>7695</td>
</tr>
<tr>
<td>3</td>
<td>(0,1,0,1)</td>
<td>6041</td>
</tr>
<tr>
<td>4</td>
<td>(1,0,0,1)</td>
<td>4273</td>
</tr>
<tr>
<td>5</td>
<td>(0,1,1,1)</td>
<td>3990</td>
</tr>
<tr>
<td>6</td>
<td>(0,0,1,1)</td>
<td>3916</td>
</tr>
<tr>
<td>7</td>
<td>(0,1,1,0)</td>
<td>3816</td>
</tr>
<tr>
<td>8</td>
<td>(0,0,1,0)</td>
<td>3647</td>
</tr>
<tr>
<td>9</td>
<td>(1,1,0,1)</td>
<td>3356</td>
</tr>
<tr>
<td>10</td>
<td>(1,1,1,1)</td>
<td>2224</td>
</tr>
<tr>
<td>11</td>
<td>(1,0,1,1)</td>
<td>2058</td>
</tr>
<tr>
<td>12</td>
<td>(1,1,0,0)</td>
<td>1282</td>
</tr>
<tr>
<td>13</td>
<td>(1,1,1,0)</td>
<td>1158</td>
</tr>
<tr>
<td>14</td>
<td>(1,0,0,0)</td>
<td>1135</td>
</tr>
<tr>
<td>15</td>
<td>(1,0,1,0)</td>
<td>905</td>
</tr>
</tbody>
</table>

configuration (see accompanying key)
<table>
<thead>
<tr>
<th>Key</th>
<th>Configuration</th>
<th>Total seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0,0,0,1)</td>
<td>14383</td>
</tr>
<tr>
<td>2</td>
<td>(0,1,0,0)</td>
<td>7695</td>
</tr>
<tr>
<td>3</td>
<td>(0,1,0,1)</td>
<td>7695</td>
</tr>
<tr>
<td>4</td>
<td>(0,1,1,0)</td>
<td>3951</td>
</tr>
<tr>
<td>5</td>
<td>(0,1,1,1)</td>
<td>3951</td>
</tr>
<tr>
<td>6</td>
<td>(0,0,1,0)</td>
<td>3647</td>
</tr>
<tr>
<td>7</td>
<td>(0,0,1,1)</td>
<td>3647</td>
</tr>
<tr>
<td>8</td>
<td>(1,0,0,0)</td>
<td>1135</td>
</tr>
<tr>
<td>9</td>
<td>(1,1,0,0)</td>
<td>1135</td>
</tr>
<tr>
<td>10</td>
<td>(1,0,1,0)</td>
<td>1135</td>
</tr>
<tr>
<td>11</td>
<td>(1,0,0,1)</td>
<td>1135</td>
</tr>
<tr>
<td>12</td>
<td>(1,1,1,0)</td>
<td>1135</td>
</tr>
<tr>
<td>13</td>
<td>(1,1,0,1)</td>
<td>1135</td>
</tr>
<tr>
<td>14</td>
<td>(1,0,1,1)</td>
<td>1135</td>
</tr>
<tr>
<td>15</td>
<td>(1,1,1,1)</td>
<td>1135</td>
</tr>
</tbody>
</table>

Table 8.7: Strongest Policy Elapsed Times
### Table 8.8: All Policies Elapsed Times

<table>
<thead>
<tr>
<th>Key</th>
<th>Configuration</th>
<th>choose_host</th>
<th>random</th>
<th>least_loaded</th>
<th>strongest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(0,0,0,1)</td>
<td>14383</td>
<td>14383</td>
<td>14383</td>
<td>14383</td>
</tr>
<tr>
<td>2</td>
<td>(0,1,0,0)</td>
<td>7695</td>
<td>7695</td>
<td>7695</td>
<td>7695</td>
</tr>
<tr>
<td>3</td>
<td>(0,1,0,1)</td>
<td>5733</td>
<td>12709</td>
<td>6041</td>
<td>7695</td>
</tr>
<tr>
<td>4</td>
<td>(0,0,1,0)</td>
<td>3647</td>
<td>3647</td>
<td>3647</td>
<td>3647</td>
</tr>
<tr>
<td>5</td>
<td>(0,1,1,0)</td>
<td>3187</td>
<td>10132</td>
<td>3916</td>
<td>3647</td>
</tr>
<tr>
<td>6</td>
<td>(0,0,1,1)</td>
<td>3025</td>
<td>6389</td>
<td>3816</td>
<td>3951</td>
</tr>
<tr>
<td>7</td>
<td>(0,1,1,1)</td>
<td>2887</td>
<td>10438</td>
<td>3990</td>
<td>3951</td>
</tr>
<tr>
<td>8</td>
<td>(1,0,0,0)</td>
<td>1135</td>
<td>1135</td>
<td>1135</td>
<td>1135</td>
</tr>
<tr>
<td>9</td>
<td>(1,0,0,1)</td>
<td>1094</td>
<td>8593</td>
<td>4273</td>
<td>1135</td>
</tr>
<tr>
<td>10</td>
<td>(1,1,0,0)</td>
<td>990</td>
<td>4942</td>
<td>1282</td>
<td>1135</td>
</tr>
<tr>
<td>11</td>
<td>(1,1,0,1)</td>
<td>986</td>
<td>9349</td>
<td>3356</td>
<td>1135</td>
</tr>
<tr>
<td>12</td>
<td>(1,0,1,0)</td>
<td>821</td>
<td>2654</td>
<td>905</td>
<td>1135</td>
</tr>
<tr>
<td>13</td>
<td>(1,0,1,1)</td>
<td>821</td>
<td>7586</td>
<td>2058</td>
<td>1135</td>
</tr>
<tr>
<td>14</td>
<td>(1,1,1,0)</td>
<td>815</td>
<td>4971</td>
<td>1158</td>
<td>1135</td>
</tr>
<tr>
<td>15</td>
<td>(1,1,1,1)</td>
<td>815</td>
<td>8498</td>
<td>2224</td>
<td>1135</td>
</tr>
</tbody>
</table>
8.5  Statistical Conclusions

The results of the simulation suggest that, in a heterogeneous configuration of computers where network and host loads are actively changing, dynamic selection of a host to execute a particular task is best made by considering the current load characteristics of all hosts and the network and extrapolating the likely elapsed time from the known network transfer time and known performance of a particular instance of the task on each host.
CHAPTER 9

Conclusion and Future Directions

9.1 Conclusion

We have explored the development and implementation of libHCS/HCSdaemon, a system for improving program execution time through dynamic task distribution in a heterogeneous, loosely-coupled distributed computer system.

The major goal of developing a cost-effective approach to heterogeneous computing with low development overhead and automated optimization has been partially satisfied. However, while this system allows users, with no additional programming effort, to execute their programs in a distributed mode, it has not been demonstrated that this system can dramatically improve the performance of those programs. The results of the clinpack benchmark only suggest that such improvement may be possible. Further study and conversion of additional subroutines to the library are indicated.

The results of our study do suggest that this is a reasonable approach. Dynamic host participation in a distributed computer has been demonstrated. It has been demonstrated that it is possible to use the same program in a non-distributed mode and a distributed mode, just by changing a library. Dynamic task-to-host binding is possible. And, there is a reasonable heuristic for making good task-to-host binding choices based
on task performance, system load, and network traffic. Further, the results of our simulation suggest that changing loads do not interfere with the usefulness of those choices.

Regarding some of the other aspects of this project:

• We have provided the motivation for such a project (see Chapter 1, Introduction.).
• We have established a theoretical basis (see Chapter 4, Abstract Model.)
• We have specified and implemented the computational model (see Chapter 5, Computational Model, and Chapter 6, Implementation.)
• We have demonstrated the feasibility of the approach through implementation of a simple prototype and through simulation (see Chapter 7, Implementation Case Study, and Chapter 8, Simulation.)
• We have demonstrated extensibility by converting a well known benchmark (see Chapter 7, Implementation Case Study.)

We have addressed the implementation issues (Chapter 6, Implementation) set forth in the introduction, to wit:

• Data portability and uniform parameter passing:
  Accomplished through the use of the ONC XDR Protocol.
• Library and daemon portability:
  Accomplished through the use of the C language and UNIX.
• Subroutine conversion and library extensibility:
  Steps have been detailed which, when followed, allow for the addition of subrou­tines to the system.
• Interprocess and interprocessor communication and synchronization:
  Accomplished through the use of the ONC RPC Protocol.
• Global state of the distributed computer:
  Accomplished through the use of cached, periodically updated, host information in libHCS routines on the local host.
• Stateless vs. stateful daemon operation:
Stateless operation was chosen.

- Metrics for determining host suitability:
  Achieved through automatic benchmarking and load statistics.
- Pathological couplings and global data:
  Addressed via the ONC XDR Protocol.

9.2 Future Directions

There has been no mention of concurrency in this thesis. This, of course, is a real area of interest. Empowering \texttt{choose\_host()} to recognize implicit parallelism in its subroutines and make choices of hosts to which to send portions of a concurrent problem would represent a real advance over the current scheme. Alternatively, and simpler, would be to use a parallelizing compiler to do this work and make concurrent calls to the libHCS routines. A new heuristic would be required for \texttt{choose\_host()}. Again, opportunistic scheduling would be desirable, since the problem of optimum distribution is NP-complete \cite{28}.

Considerable work could also be done on the present implementation, for example:

- Bypass network code when executing on the local host.
- The use of benchmark results from a single instance of an algorithm is not always representative of the performance potential, in all instances, of a particular architecture. Sometimes larger instances perform, relatively, better than small on some architectures, and vice versa.
- Automated subroutine conversion.

9.3 Final Thoughts

Heterogeneous computing will continue to be an active research area, since there is no theoretical basis to a hope that there exists a 'Unified Computational Theory' that
allows for the selection of a single, best, architecture to compute the diversity of computational algorithms. In addition, problems associated with the concurrent execution of program elements across a heterogeneous collection of loosely-connected computers, an area of great practical interest, will experience an upsurge as the sequential programming model reaches its theoretical, architectural limits.
APPENDIX A

libHCS.x

/*
 * libHCS.x
 *
 * This is the libHCS protocol file for the rpcgen(1) protocol
 * compiler. Compiling this file with rpcgen(1) generates files:
 *
 * libHCS.h
 * libHCS_clnt.c
 * libHCS_xdr.c
 * libHCS_svc.c
 *
 * In addition to these files, the two files:
 *
 * libHCS.c
 * libHCS_l.c
 *
 * are required to build libHCS/HCSdaemon.
 *
 * libHCS is comprised of:
 *
 * libHCS.h
 * (plus any subroutine-specific header files)
 * libHCS.c
 * libHCS_clnt.c
 * libHCS_xdr.c
 *
 * While HCSdaemon is comprised of:
 *
 * libHCS.h
 * (plus any subroutine-specific header files)
 * libHCS_svc.c
 * libHCS_xdr.c
 * libHCS_l.c
 *
 * Note: Adding an entry to libHCS.x is one of the steps to adding
 * a new library routine to libHCS/HCSdaemon. The following
 * instructions describe the process of adding an entry, func(),
 * to libHCS.x:
 */
1) Collect the argument list into a single C structure. Include any global variables that must be passed. Name the structure - func_arg.

2) Collect all variables used to return values to the calling program into another C structure. Duplicate any pass-by-reference items appearing in the argument list, as well as any global variables, and the function's original return value. Name the structure - func_res.

3) Add a new procedure specification:

   func_res FUNC(func_arg) = #;

   where '#' is the next procedure number in the list.

Following these additions to libHCS.x, and in a safe place away from the production libHCS/HCSdaemon code, compile the modified libHCS.x file with rpcgen. Then:

4) Use libHCS.h as it comes from rpcgen.

5) Extract the relevant entries from the libHCS_clnt.c file and include them in the proper place in the production libHCS_clnt.c file (see the production libHCS_clnt.c).

6) Extract the relevant entries from the libHCS_svc.c file and include them in the proper place in the production libHCS_svc.c file (see the production libHCS_xdr.c).

7) Use libHCS_xdr.c as it comes from rpcgen.

8) Rebuild the system.

const MAXROUTINES = 10;
const MAXHOSTS = 30;
const MAXHOSTLEN = 100;
const MAXORDER = 20;
const POLL_TIME = 30;
const O_1 = 1;
const O_N = 2;
const O_NLogN = 3;
const O_N2 = 4;
const O_N3 = 5;

struct benchmark {
    long routine;
    int size;
    int order;
    float time;
};
struct eight_k_arg {
    int array[8192];
};

struct eight_k_res {
    int array[8192];
};

struct gethost_arg {
    u_long routine;
};

struct gethost_res {
    int load;
    int rcnt;
    struct benchmark bench[MAXROUTINES];
};

struct addf_arg {
    float x;
    float y;
};

struct addf_res {
    float z;
};

struct mulf_arg {
    float x;
    float y;
};

struct mulf_res {
    float z;
};

struct divf_arg {
    float x;
    float y;
};

struct divf_res {
    float z;
};

struct daxpy_arg {
    int n;
    float da;
    float dx<>
    int incx;
    float dy<>
    int incy;
};
struct daxpy_res {
    float dy<>;
};

struct dgesl_arg {
    float a<>;
    int lda;
    int n;
    int ipvt<>;
    float b<>;
    int job;
};

struct dgesl_res {
    float b<>;
};

struct dgefa_arg {
    float a<>;
    int lda;
    int n;
    int ipvt<>;
    int *info;
};

struct dgefa_res {
    float a<>;
    int ipvt<>;
    int *info;
};

program HCSDAEMON {
    version HCSVERS {
        eight_k_res EIGHT_K(eight_k_arg) = 1002;
        gethost_res GETHOST(void) = 1001;
        addf_res ADDF(addf_arg) = 1;
        mulf_res MULF(mulf_arg) = 2;
        divf_res DIVF(divf_arg) = 3;
        daxpy_res DAXPY(daxpy_arg) = 4;
        dgesl_res DGESL(dgesl_arg) = 5;
        dgefa_res DGEFA(dgefa_arg) = 6;
    } = 1;
} = 0x20000100;

#if RPC_HDR
%
%struct host {
%    char name[MAXHOSTLEN];
%    int valid;
%    int load;
%    CLIENT *clnt_handlep;
%    float network_latency;
%    float xfer_latency;
%    float gethost_res {
%        int lda;
%        int n;
%        int ipvt<>;
%        float b<>;
%        int job;
%    };
%    float daxpy_res {
%        float dy<>;
%    };
%    float dgesl_res {
%        float b<>;
%    };
%    float dgefa_res {
%        float a<>;
%        int ipvt<>;
%        int *info;
%    };
%    version HCSVERS {
%        eight_k_res EIGHT_K(eight_k_arg) = 1002;
%        gethost_res GETHOST(void) = 1001;
%        addf_res ADDF(addf_arg) = 1;
%        mulf_res MULF(mulf_arg) = 2;
%        divf_res DIVF(divf_arg) = 3;
%        daxpy_res DAXPY(daxpy_arg) = 4;
%        dgesl_res DGESL(dgesl_arg) = 5;
%        dgefa_res DGEFA(dgefa_arg) = 6;
%    } = 1;
%}
#endif
% int rcnt;
% struct benchmark bench[MAXROUTINES];
% float score[MAXROUTINES];
%
#endif

#endif
APPENDIX B

libHCS

/*
 * libHCS.h
 *
 * This header file is used by libHCS.a and HCSdaemon.
 * It contains #defines for some of the basic data structure
 * sizes. It contains all XDR data type specifications.
 * Finally, it contains the basic RPC definition of the
 * libHCS/HCSdaemon protocol, as well as procedure IDs
 * for all RPCable procedures.
 */
#define MAXROUTINES 10
#define MAXHOSTS 30
#define MAXHOSTLEN 100
#define MAXORDER 20
#define POLL_TIME 30
#define O_1 1
#define O_N 2
#define O_NLogN 3
#define O_N2 4
#define O_N3 5

struct benchmark {
    long routine;
    int size;
    int order;
    float time;
};
typedef struct benchmark benchmark;
bool_t xdr_benchmark();

struct eight_k_arg {
    int array[8192];
};
typedef struct eight_k_arg eight_k_arg;
bool_t xdr_eight_k_arg();
struct eight_k_res {
    int array[8192];
};
typedef struct eight_k_res eight_k_res;
bool_t xdr_eight_k_res();

struct gethost_arg {
    u_long routine;
};
typedef struct gethost_arg gethost_arg;
bool_t xdr_gethost_arg();

struct gethost_res {
    int load;
    int rcnt;
    struct benchmark bench[MAXROUTINES];
};
typedef struct gethost_res gethost_res;
bool_t xdr_gethost_res();

struct addf_arg {
    float x;
    float y;
};
typedef struct addf_arg addf_arg;
bool_t xdr_addf_arg();

struct addf_res {
    float z;
};
typedef struct addf_res addf_res;
bool_t xdr_addf_res();

struct mulf_arg {
    float x;
    float y;
};
typedef struct mulf_arg mulf_arg;
bool_t xdr_mulf_arg();

struct mulf_res {
    float z;
};
typedef struct mulf_res mulf_res;
bool_t xdr_mulf_res();

struct divf_arg {
float x;
float y;
};
typedef struct divf_arg divf_arg;
bool_t xdr_divf_arg();

struct divf_res {
    float z;
};
typedef struct divf_res divf_res;
bool_t xdr_divf_res();

struct daxpy_arg {
    int n;
    float da;
    struct {
        u_int dx_len;
        float *dx_val;
    } dx;
    int incx;
    struct {
        u_int dy_len;
        float *dy_val;
    } dy;
    int incy;
};
typedef struct daxpy_arg daxpy_arg;
bool_t xdr_daxpy_arg();

struct daxpy_res {
    struct {
        u_int dy_len;
        float *dy_val;
    } dy;
};
typedef struct daxpy_res daxpy_res;
bool_t xdr_daxpy_res();

struct dgesl_arg {
    struct {
        u_int a_len;
        float *a_val;
    } a;
    int lda;
    int n;
    struct {
        u_int ipvt_len;
        int *ipvt_val;
    } ipvt;
    struct {

u_int b_len;
    float *b_val;
    int job;
};
typedef struct dgesl_arg dgesl_arg;
bool_t xdr_dgesl_arg();

struct dgesl_res {
    struct {
        u_int b_len;
        float *b_val;
    } b;
};
typedef struct dgesl_res dgesl_res;
bool_t xdr_dgesl_res();

struct dgefa_arg {
    struct {
        u_int a_len;
        float *a_val;
    } a;
    int lda;
    int n;
    struct {
        u_int ipvt_len;
        int *ipvt_val;
    } ipvt;
    int *info;
};
typedef struct dgefa_arg dgefa_arg;
bool_t xdr_dgefa_arg();

struct dgefa_res {
    struct {
        u_int a_len;
        float *a_val;
    } a;
    struct {
        u_int ipvt_len;
        int *ipvt_val;
    } ipvt;
    int *info;
};
typedef struct dgefa_res dgefa_res;
bool_t xdr_dgefa_res();

#define HCSDAEMON ((u_long)0x20000100)
#define HCSVERS ((u_long)1)
#define EIGHT_K ((u_long)1002)
extern eight_k_res *eight_k_l();
#define GETHOST ((u_long)1001)
extern gethost_res *gethost_l();
#define ADDF ((u_long)1)
extern addf_res *addf_l();
#define MULF ((u_long)2)
extern mulf_res *mulf_l();
#define DIVF ((u_long)3)
extern divf_res *divf_l();
#define DAXPY ((u_long)4)
extern daxpy_res *daxpy_l();
#define DGESL ((u_long)5)
extern dgesl_res *dgesl_l();
#define DGEFA ((u_long)6)
extern dgefa_res *dgefa_l();

struct host {
    char name[MAXHOSTLEN];
    int valid;
    int load;
    CLIENT *clnt_handlep;
    float network_latency;
    float xfer_latency;
    int rcnt;
    struct benchmark bench[MAXROUTINES];
    float score[MAXROUTINES];
};
libHCS.c

libHCS.c is the bridge between the user calls to standard routines and the client-side interface to the network versions of those routines. It contains subroutine stubs which are called and return exactly like the corresponding standard routine. This extra calling level has been established so that the output from rpcgen can be used, with architecture-specific exceptions, as-is. Hopefully, any execution penalty incurred for the extra calling level will be outweighted by the execution time of the individual calls.

libHCS.c contains the logic for deciding which remote host to send the request to. So the subroutine stubs have calls to choose_host().

Argument- and return-value conversions are performed within the subroutine stubs in this source file.

#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/time.h>
#include <math.h>

libHCS.h is the .h output from the rpcgen program (see the libHCS protocol specification, libHCS.x, for more)

#include "libHCS.h"

The sample library was chosen from the C version of the linpack benchmark. This library has several #defines, etc., which need to be included here. In general, when including routines from other libraries, their include files will have to be dealt with here.

#include "clinpack.h"

Local functions

float estimate_exec();
float estimate_xfer_latency();

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

Note: Adding an entry to libHCS.c is one of the steps to adding a new library routine to libHCS. The following instructions describe the process of adding an entry to libHCS.c.

1) Duplicate the original function's prototype
2) Add the declarations, for function ‘func’:

```c
CLIENT *clnt_handlep;
int choose_host();
func_arg arg;
func_res res;
```

3) Copy arguments to the routine’s ‘arg’ data structure

4) Add a call to ‘choose_host’:

```c
if (choose_host(FUNC, prob_size, arg_size,
                res_size, &clnt_handlep) == 1) {
    fprintf(stderr,"func: choose_host failed\n");
    exit(1);
}
```

The arguments provided to this call assume the programmer has some basic knowledge about the current invocation -

- prob_size = current ‘size’ of the problem (for estimation of its complexity.
- arg_size = size of argument data to be sent.
- res_size = size of results data to be returned.

5) Call the network version of the routine with the returned client handle

```c
res = *func_l(&arg, clnt_handlep);
```

6) Copy, if necessary, any return values to local variables.

7) Free, if necessary, any result arrays implicitly allocated by the XDR routines.

8) Return, if necessary, the value of the function.

In addition to the foregoing, entries must be made in the following files:

- libHCS.h (and, by extension, libHCS.x)
  Data structures needed by the routine, as well as XDR definitions.
- libHCS_clnt.c
  The client-side network call to invoke the subroutine on a remote host.
- libHCS_svc.c
The server-side code to receive the call.

- **libHCS_xdr.c**
  - The XDR routines necessary to encode and decode, in a host-independent format, the data structures needed by the routine.

- **libHCS_l.c**
  - The actual subroutine code to be executed by the remote host.

/*
 * addf(), mulf() and divf() were test routines which were used to establish the feasibility of this approach. They have been left in because their simplicity aids illustration without obscuring the underlying technique.
 */

float addf(float x, float y)
{
    CLIENT *clnt_handlep;
    int choose_host();
    addf_arg arg;
    addf_res res;

    arg.x = x;
    arg.y = y;

    if (choose_host(ADDF, 1, sizeof(arg), sizeof(res), &clnt_handlep) == 1) {
        fprintf(stderr, "addf: choose_host failed\n");
        exit(1);
    }

    res = *addf_l(&arg, clnt_handlep);

    return(res.z);
}

float mulf(float x, float y)
{
    CLIENT *clnt_handlep;
    mulf_arg arg;
    mulf_res res;

    arg.x = x;
    arg.y = y;

    if (choose_host(MULF, 1, sizeof(arg), sizeof(res), &clnt_handlep) == 1) {
        fprintf(stderr, "mulf: choose_host failed\n");
        exit(1);
    }

    res = *mulf_l(&arg, clnt_handlep);
return(res.z);
}

float divf(float x, float y) 
{
    CLIENT *clnt_handlep;
    divf_arg arg;
    divf_res res;

    arg.x = x;
    arg.y = y;

    if (choose_host(DIVF,1,sizeof(arg),sizeof(res),&clnt_handlep) == 1) {
        fprintf(stderr,"divf: choose_host failed\n");
        exit(1);
    }

    res = *divf_l(&arg,clnt_handlep);

    return(res.z);
}

/*@ */
int daxpy(n,da,dx,incx,dy,incy) 
REAL dx[],dy[],da;
int incx,incy,n;
{
    CLIENT *clnt_handlep;
    daxpy_arg arg;
    daxpy_res res;
    int i;

    arg.n = n;
    arg.da = da;
    arg.dx.dx_val = dx;
    arg.dx.dx_len = n;
    arg.incx = incx;
    arg.dy.dy_val = dy;
    arg.dy.dy_len = n;
    arg.incy = incy;

    if (choose_host(DAXPY,
            n,sizeof(arg),sizeof(res),&clnt_handlep) == 1) {
        fprintf(stderr,"daxpy: choose_host failed\n");
        exit(1);
    }

    res = *daxpy_l(&arg,clnt_handlep);

    for (i = 0; i < n; i++)
        dy[i] = res.dy.dy_val[i];
if (!clnt_freeres(clnt_handlep, xdr_daxpy_res, (char *)&res))
    fprintf(stderr,"freeres, failed, res = %d\n",res);
return;
*/

int dgesl(a, lda, n, ipvt, b, job)
int lda, n, ipvt[], job;
REAL a[], b[];
{
    CLIENT *clnt_handlep;
    dgesl_arg arg;
    dgesl_res res;
    int i;
    arg.a.a_val = a;
    arg.a.a_len = n * lda;
    arg.lda = lda;
    arg.n = n;
    arg.ipvt.ipvt_val = ipvt;
    arg.ipvt.ipvt_len = n;
    arg.b.b_val = b;
    arg.b.b_len = n;
    arg.job = job;

    if (choose_host(DGESL,n,
                    sizeof(arg) + 4*(arg.a.a_len+arg.ipvt.ipvt_len+arg.b.b_len),
                    sizeof (res) + 4*n, &clnt_handlep) == 1) {
        fprintf(stderr,"dgesl:  choose_host failed\n");
        exit(1);
    }

    res = *dgesl_l(&arg,clnt_handlep);

    for (i = 0; i < n; i++)
        b[i] = res.b.b_val[i];

    if (!clnt_freeres(clnt_handlep,xdr_dgesl_res,(char *)&res))
        fprintf(stderr,"freeres, failed, res = %d\n",res);
    return;
}

int dgefa(a, lda, n, ipvt, info)
REAL a[];
int lda, n, ipvt[], *info;
{
    CLIENT *clnt_handlep;
    dgefa_arg arg;
    dgefa_res res;
    int i;
    arg.a.a_val = a;
    arg.a.a_len = n * lda;
    arg.lda = lda;
arg.n = n;
arg.ipvt.ipvt_val = ipvt;
arg.ipvt.ipvt_len = n;
arg.info = info;

if (choose_host(DGEFA,n,
    sizeof(arg) + 4*(arg.a.a_len+arg.ipvt.ipvt_len),
    sizeof(res) + 4*((n*lda)+n), &clnt_handlep) == 1) {
    fprintf(stderr,"dgefa: choose_host failed\n");
    exit(1);
}

res = *dgefa_l(&arg,clnt_handlep);

for (i = 0; i < (n * lda); i++)
    a[i] = res.a.a_val[i];

for (i = 0; i < n; i++)
    ipvt[i] = res.ipvt.ipvt_val[i];

*info = *res.info;

if (!clnt_freeres(clnt_handlep,xdr_dgefa_res, (char *)&res))
    fprintf(stderr,"freeres, failed, res = %d\n", res);
return;

/*
 * Function choose_host() is the key element of the libHCS system.
 * choose_host() is called by each function, func(x), in the library
 * to discover the best host to execute the current instance of
 * func(x).
 *
 * The following steps are performed by choose_host:
 * 1) Discover which hosts will be participating (done the first
 *    time through.)
 * 2) Establish RPC communication with each host.
 * 3) Poll, at POLL_TIME intervals, each available host to
 *    discover its load as well as network latency to the host.
 * 4) Acquire from each available host the benchmark data for each
 *    routine.
 * 5) Calculate the score for each routine called (estimated time
 *    to: a) transfer data, b) execute the routine on the current
 *    input and c) and return results) for each host.
 * 6) Pick the host with the lowest score and return its RPC
 *    handle.
 */
int choose_host(u_long routine, int size,
int sizeof_arg, int sizeof_res, CLIENT **clnt)
{
    static struct host hosts[MAXHOSTS];
    static int first_pass = 0;
    static clock_t watchdog_timer = (clock_t)0;
    clock_t temp_watchdog;
    struct tms dummy;

    float xfer_ratio;
    float send_latency;
    float recv_latency;

    void *gethost_arg = NULL;
    gethost_res gethost_res;

    struct timeval tp;
    struct timeval tv;
    struct timezone tz; /* leftover */

    char *hostlist[MAXHOSTS];
    static int hostcnt = 0;

    eight_k_arg eight_k_arg;
    eight_k_res eight_k_res;

    int i,j;

    /*
    * The first time through, we need to find out which hosts will be
    * participating. Currently, this will be limited to a simple list
    * read from the file hostlist.
    */
    if (first_pass == 0) {
        first_pass = 1;
        /*
        * Generate a list of available hosts. Currently, we only do
        * this one time. Ideally, as hosts come and go on the
        * network (if, indeed, the network is that dynamic) we should
        * be able to adjust.
        */
        hostcnt = gethostlist(hostlist);
    }

    /*
    * Find competent hosts - hosts which are running HCSdaemon.
    * Don't worry about failed clnt_create calls. If the handle is
    * NULL we just don't use it (i.e. the host is incompetent.)
    */
    for (i = 0, j = 0; i < hostcnt; i++, j++) {
        strcpy(hosts[j].name,hostlist[i]);
        hosts[j].clnt_handlep =
            clnt_create(hosts[j].name, HCSDAEMON, HCSVERS, "tcp");
        if (hosts[j].clnt_handlep == (CLIENT *)NULL) {
            clnt_pcreateerror(hosts[j].name);
        }
else
    hosts[j].valid = 0;
}
hostcnt = j;
}

/*
 * Check our watchdog timer to see if it's time to re-poll (i.e. to
 * see if POLL_TIME seconds have elapsed since the last time we
 * polled.) If so, then just invalidate all available hosts. This
 * will force the scoring code below to be executed which does the
 * poll.
 */

temp_watchdog = times(&dummy);
if (((temp_watchdog - watchdog_timer)/(float)CLK_TCK) >
    (float)POLL_TIME) {
    int j;
    watchdog_timer = temp_watchdog;
    for (j = 0; j < hostcnt; j++)
        hosts[j].valid = 0;
}

/*
 * Now go through the list and discover the best candidate for this
 * task. This process makes the following assumptions:
 * o Each host reports its load as a percentage of 'available'
 *     processing potential (including the current function
 *     execution, if selected.)
 * o Network latency can be discovered with a simple NULLPROC
 *     RPC call.
 * o Transfer latency can be discovered by transferring an 8KB
 *     buffer. This latency value can then be used to estimate
 *     the time required to send arguments and return results.
 * o Benchmark data is present, per host, for all callable
 *     routines, which includes size and time for the benchmark
 *     execution as well as the order of execution complexity.
 *     This information can be used to estimate the execution
 *     time of an instance of the routine with size size'.
 */
for (i = 0; i < hostcnt; i++)
    /*
     * If we haven't already gotten load and performance information
     * from server i, do so now.
     */
    if (hosts[i].valid == 0) {
        /*
         * Make sure we can get to this host and, as a side effect,
         * discover its network latency (via a NULLPROC call).
         */
*/
gettimeofday( &tp, &tz );
if (clnt_call(hosts[i].clnt_handlep, NULLPROC,
    xdr_void, 0, xdr_void, 0, TIMEOUT) != RPC_SUCCESS) {
    clnt_error(hosts[i].clnt_handlep, hosts[i].name);
    hosts[i].valid = 0;
    break;
} else
    hosts[i].valid = 1;

gettimeofday( &tv, &tz );
tvsub( &tv, &tp );
hosts[i].network_latency =
    tv.tv_sec+((float)tv.tv_usec/1000000.0);

/*
 *  Next, discover the 8KB transfer latency to this host.
 */
gettimeofday( &tp, &tz );
eight_k_res = *eight_k_l(&eight_k_arg,hosts[i].clnt_handlep);
gettimeofday( &tv, &tz );
tvsub( &tv, &tp );
hosts[i].xfer_latency =
    (tv.tv_sec+((float)tv.tv_usec/1000000.0))/2.0;

/*
 *  Finally, get this host’s load and the benchmark data for
 *  all routines known to it. Then invalidate this host’s
 *  scores for all routines.
 */
gethost_res = *gethost_l(gethost_arg,hosts[i].clnt_handlep);
hosts[i].load = gethost_res.load;
hosts[i].rcnt = gethost_res.rcnt;
for (j = 0; j < hosts[i].rcnt; j++) {
    hosts[i].bench[j] = gethost_res.bench[j];
    hosts[i].score[j] = -1;
}

/*
 *  OK, calculate, if necessary, each server’s score for the current
 *  routine
 */
for (i = 0; i < hostcnt; i++)
    if (hosts[i].valid == 1)
        if (routine >= hosts[i].rcnt) {
            fprintf(stderr,"internal error\n");
            fprintf(stderr,"marking %s as invalid\n",hosts[i].name);
            hosts[i].valid = 0;
        }
    else
        /*
        *  The score calculation is the sum of the send latency,
* execution time, and receive latency.
*/
if (hosts[i].score[routine] == -1) {
    hosts[i].score[routine] =
        estimate_xfer_latency(sizeof_arg, &hosts[i]);
    hosts[i].score[routine] +=
        estimate_exec(routine, &hosts[i], size);
    hosts[i].score[routine] +=
        estimate_xfer_latency(sizeof_res, &hosts[i]);
}

/*
 * Pick the best score (lowest time) from all hosts.
*/
*clnt = hosts[hselect(hostcnt, hosts, routine)].clnt_handlep;

return(0);
}

/*
 * hselect()
 *
 * Routine to go through the list of available hosts and pick the one
 * with the lowest score.
 */
int hselect(int hostcnt, struct host hosts[], long routine)
{
    int computed_index[10];
    int i, minindex;
    float min;

    min = 99999999.9;
    minindex = 0;
    for (i = 0; i < hostcnt; i++)
        if (hosts[i].valid && (hosts[i].score[routine] < min)) {
            min = hosts[i].score[routine];
            minindex = i;
        }
    return(minindex);
}

/*
 * gethostlist()
 *
 * Routine to produce a list of candidate hosts for libHCS.
 *
 * (This should be more generic... however, and for example, using
 * something like broadcast_rpc, which doesn't even work on the Cray,
 * would only discover locally connected hosts. So, for now, just
 * read in a list of hosts.)
 */
int gethostlist(char *hostlist[])
{
    static char lhostlist[MAXHOSTS][MAXHOSTLEN];
    int i;
int hcnt;
char fname[MAXHOSTLEN];
FILE *fd;

fd = fopen("hostlist","r");

hcnt = 0;
if (fd) {
    while ((hcnt < MAXHOSTS) && (fscanf (fd,"%s",fname) != EOF))
        strcpy (lhostlist[hcnt++],fname);
    fclose(fd);
} else
    strcpy(lhostlist[hcnt++],"localhost");

for (i = 0; i < hcnt; i++)
    hostlist[i] = lhostlist[i];

return (hcnt);

/*
 * TVSUB
 *
 * Subtract 2 timeval structs: out = out - in.
 *
 * Out is assumed to be >= in.
 */
tvsub(out, in)
     register struct timeval *out, *in;
{
    if( (out->tv_usec -= in->tv_usec) < 0 ) {  
        out->tv_sec--;  
        out->tv_usec += 1000000;
    }
    out->tv_sec -= in->tv_sec;
}

/*
 * estimate_exec
 *
 * This routine uses the benchmark information acquired from each
 * participating host, as well as the size of the current routine,
 * to estimate the execution time of this instance of the routine.
 */
float estimate_exec(int routine, struct host *hostp, int size)
{
    float size_ratio;
    float time_ratio;
    float exec_time;

    /*
    * Assume valid numbers
    */
    size_ratio = (float)size/(float)hostp->bench[routine].size;
switch (hostp->bench[routine].order) {
    case 0_1:
        time_ratio = 1;
        break;
    /*
      * linear relationship => n'/n
    */
    case 0_N:
        time_ratio = size_ratio;
        break;
    /*
      * nlogn case => n'*log(n')/n*log(n) =>
      * size_ratio * log(n')/log(n)
    */
    case 0_NLogN:
        time_ratio = size_ratio * log(size)/log(hostp->bench[routine].size);
        break;
    /*
      * n^2 case => n'^2/n^2 => (n' * n')/(n * n) =>
      * size_ratio * size_ratio
      * (same for n^3)
    */
    case 0_N2:
        time_ratio = size_ratio * size_ratio;
        break;
    case 0_N3:
        time_ratio = size_ratio * size_ratio * size_ratio;
        break;
}
exec_time = (time_ratio * hostp->bench[routine].time)/
            (((float)hostp->load)/100.0);
return(exec_time);

/*
 * estimate_xfer_latency
 * This routine uses the transfer latency acquired from each
 * participating host, as well as the size of the current routine’s
 * data transfer to estimate the transfer time for this instance of
 * the routine.
 */
float estimate_xfer_latency(int xfer_size, struct host *hostp) {
    float xfer_ratio;
    float latency;

    xfer_ratio = ((float)(xfer_size))/8192.0;
    latency = hostp->network_latency;
    latency+= (hostp->network_latency < hostp->xfer_latency)?
              xfer_ratio * (hostp->xfer_latency - hostp->network_latency) :

xfer_ratio * hostp->xfer_latency;

return(latency);
#include <rpc/rpc.h>
#include <sys/time.h>
#include "libHCS.h"

static struct timeval TIMEOUT = { 25, 0 };  

 eight_k_res *
 eight_k_l(argp, clnt)
     eight_k_arg *argp;
     CLIENT *clnt;
 {
 static eight_k_res res;

 (void) memset((char *)&res, 0, sizeof (res));

 if (clnt_call(clnt, EIGHT_K, xdr_eight_k_arg, argp, 
 xdr_eight_k_res, &res, TIMEOUT) != RPC_SUCCESS) {
 clnt_perror(clnt,"eight_k_l");
 return (NULL);
return (&res);
}

gethost_res *
gethost_l(argp, clnt)
void *argp;
CLIENT *clnt;
{
    static gethost_res res;
    (void) memset((char *)&res, 0, sizeof (res));

    if (clnt_call(clnt, GETHOST, xdr_void, argp,
       xdr_gethost_res, &res, TIMEOUT) != RPC_SUCCESS) {
        clnt_perror(clnt,"gethost_l");
        return (NULL);
    }
    return (&res);
}

addf_res *
addf_l(argp, clnt)
addf_arg *argp;
CLIENT *clnt;
{
    static addf_res res;
    (void) memset((char *)&res, 0, sizeof (res));

    if (clnt_call(clnt, ADDF, xdr_addf_arg, argp,
       xdr_addf_res, &res, TIMEOUT) != RPC_SUCCESS) {
        clnt_perror(clnt,"addf_l");
        return (NULL);
    }
    return (&res);
}

mulf_res *
mulf_l(argp, clnt)
mulf_arg *argp;
CLIENT *clnt;
{
    static mulf_res res;
    (void) memset((char *)&res, 0, sizeof (res));

    if (clnt_call(clnt, MULF, xdr_mulf_arg, argp,
       xdr_mulf_res, &res, TIMEOUT) != RPC_SUCCESS) {
        clnt_perror(clnt,"mulf_l");
        return (NULL);
    }
    return (&res);
}
divf_res *
divf_l(argp, clnt)
divf_arg *argp;
CLIENT *clnt;
{
    static divf_res res;
    (void) memset((char *)&res, 0, sizeof (res));
    if (clnt_call(clnt, DIVF, xdr_divf_arg, argp,
                  xdr_divf_res, &res, TIMEOUT) != RPC_SUCCESS)
    {
        clnt_perror(clnt,"divf_l");
        return (NULL);
    }
    return (&res);
}

daxpy_res *
daxpy_l(argp, clnt)
daxpy_arg *argp;
CLIENT *clnt;
{
    static daxpy_res res;
    (void) memset((char *)&res, 0, sizeof (res));
    if (clnt_call(clnt, DAXPY, xdr_daxpy_arg, argp,
                  xdr_daxpy_res, &res, TIMEOUT) != RPC_SUCCESS)
    {
        return (NULL);
    }
    return (&res);
}

dgesl_res *
dgesl_l(argp, clnt)
dgesl_arg *argp;
CLIENT *clnt;
{
    static dgesl_res res;
    (void) memset((char *)&res, 0, sizeof (res));
    if (clnt_call(clnt, DGESL, xdr_dgesl_arg, argp,
                  xdr_dgesl_res, &res, TIMEOUT) != RPC_SUCCESS)
    {
        return (NULL);
    }
dgesfa_res *
dgesfa_l(argp, clnt)
    dgesfa_arg *argp;
    CLIENT *clnt;
{
    static dgesfa_res res;

    (void) memset((char *)&res, 0, sizeof (res));

    if (clnt_call(clnt, DGEFA, xdr_dgesfa_arg, argp,
        xdr_dgesfa_res, &res, TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&res);
}
/ * libHCS  libHCS_xdr.c
 * libHCS
 * libHCS This file is used, unchanged (except for these comments),
 * libHCS as generated by rpcgen(1) via:
 * libHCS
 * libHCS    rpcgen libHCS.x
 * libHCS
 * libHCS This file implements the XDR (eXternal Data Representation)
 * libHCS standard.
 */
#include <rpc/rpc.h>
#include "libHCS.h"

bool_t
xdr_benchmark(xdrs, objp)
    XDR *xdrs;
    benchmark *objp;
{
    if (!xdr_long(xdrs, &objp->routine)) {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->size)) {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->order)) {
        return (FALSE);
    }
    if (!xdr_float(xdrs, &objp->time)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_eight_k_arg(xdrs, objp)
    XDR *xdrs;
    eight_k_arg *objp;
{
    if (!xdr_vector(xdrs,
        (char *)objp->array, 8192, sizeof(int), xdr_int)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_eight_k_res(xdrs, objp)
XDR *xdrs;
eight_k_res *objp;
{
    if (!xdr_vector(xdrs, (char *)objp->array, 8192, sizeof(int), xdr_int)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_gethost_arg(xdrs, objp)
    XDR *xdrs;
    gethost_arg *objp;
{
    if (!xdr_u_long(xdrs, &objp->routine)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_gethost_res(xdrs, objp)
    XDR *xdrs;
    gethost_res *objp;
{
    if (!xdr_int(xdrs, &objp->load)) {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->rcnt)) {
        return (FALSE);
    }
    if (!xdr_vector(xdrs, (char *)objp->bench, MAXROUTINES, sizeof(benchmark), xdr_benchmark)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_addf_arg(xdrs, objp)
    XDR *xdrs;
    addf_arg *objp;
{
    if (!xdr_float(xdrs, &objp->x)) {

bool_t
xdr_addf_res(xdrs, objp)
XDR *xdrs;
addf_res *objp;
{
    if (!xdr_float(xdrs, &objp->z)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_mulf_arg(xdrs, objp)
XDR *xdrs;
mulf_arg *objp;
{
    if (!xdr_float(xdrs, &objp->x)) {
        return (FALSE);
    }
    if (!xdr_float(xdrs, &objp->y)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_mulf_res(xdrs, objp)
XDR *xdrs;
mulf_res *objp;
{
    if (!xdr_float(xdrs, &objp->z)) {
        return (FALSE);
    }
    return (TRUE);
}
bool_t
xdr_divf_arg(xdrs, objp)
    XDR *xdrs;
    divf_arg *objp;
    {
        if (!xdr_float(xdrs, &objp->x)) {
            return (FALSE);
        }
        if (!xdr_float(xhrs, &objp->y)) {
            return (FALSE);
        }
        return (TRUE);
    }

bool_t
xdr_divf_res(xdrs, objp)
    XDR *xhrs;
    divf_res *objp;
    {
        if (!xdr_float(xhrs, &objp->z)) {
            return (FALSE);
        }
        return (TRUE);
    }

bool_t
xdr_daxpy_arg(xhrs, objp)
    XDR *xhrs;
    daxpy_arg *objp;
    {
        if (!xdr_int(xhrs, &objp->n)) {
            return (FALSE);
        }
        if (!xdr_float(xhrs, &objp->da)) {
            return (FALSE);
        }
        if (!xdr_array(xhrs, (char **)&objp->dx.dx_val,
                        (u_int *)&objp->dx.dx_len, -0, sizeof(float), xdr_float)) {
            return (FALSE);
        }
        if (!xdr_int(xhrs, &objp->incx)) {
            return (FALSE);
        }
        if (!xdr_array(xhrs, (char **)&objp->dy.dy_val,
                        (u_int *)&objp->dy.dy_len, -0, sizeof(float), xdr_float)) {
            return (FALSE);
        }
if (!xdr_int(xdrs, &objp->incy)) {
    return (FALSE);
}
return (TRUE);

bool_t
xdr_daxpy_res(xdrs, objp)
XDR *xdrs;
daxpy_res *objp;
{
    if (!xdr_array(xdrs, (char **)&objp->dy.dy_val,
        (u_int *)&objp->dy.dy_len, ~0, sizeof(float), xdr_float)) {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_dgesl_arg(xdrs, objp)
XDR *xdrs;
dgesl_arg *objp;
{
    if (!xdr_array(xdrs, (char **)&objp->a.a_val,
        (u_int *)&objp->a.a_len, ~0, sizeof(float), xdr_float)) {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->lda)) {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->n)) {
        return (FALSE);
    }
    if (!xdr_array(xdrs, (char **)&objp->ipvt.ipvt_val,
        (u_int *)&objp->ipvt.ipvt_len, ~0, sizeof(int), xdr_int)) {
        return (FALSE);
    }
    if (!xdr_array(xdrs, (char **)&objp->b.b_val,
        (u_int *)&objp->b.b_len, ~0, sizeof(float), xdr_float)) {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->job)) {
        return (FALSE);
    }
    return (TRUE);
}
bool_t
xdr_dgesl_res(xdrs, objp)
    XDR *xdrs;
    dgesl_res *objp;
{
    if (!xdr_array(xdrs, (char **)&objp->b.b_val,
                   (u_int *)&objp->b.b_len, ~0, sizeof(float), xdr_float))
    {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_dgefa_arg(xdrs, objp)
    XDR *xdrs;
    dgefa_arg *objp;
{
    if (!xdr_array(xdrs, (char **)&objp->a.a_val,
                   (u_int *)&objp->a.a_len, ~0, sizeof(float), xdr_float))
    {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->lda))
    {
        return (FALSE);
    }
    if (!xdr_int(xdrs, &objp->n))
    {
        return (FALSE);
    }
    if (!xdr_array(xdrs, (char **)&objp->ipvt.ipvt_val,
                   (u_int *)&objp->ipvt.ipvt_len, ~0, sizeof(int), xdr_int))
    {
        return (FALSE);
    }
    if (!xdr_pointer(xdrs, (char **)&objp->info, sizeof(int), xdr_int))
    {
        return (FALSE);
    }
    return (TRUE);
}

bool_t
xdr_dgefa_res(xdrs, objp)
    XDR *xdrs;
    dgefa_res *objp;
{
    if (!xdr_array(xdrs, (char **)objp->a.a_val,
                   (u_int *)objp->a.a_len, ~0, sizeof(float), xdr_float))
    {
        return (FALSE);
    }
}
if (!xdr_array(xdrs, (char **)&objp->ipvt.ipvt_val,
   (u_int *)&objp->ipvt.ipvt_len, ~0, sizeof(int), xdr_int)) {
    return (FALSE);
}
if (!xdr_pointer(xdrs, (char **)&objp->info, sizeof(int), xdr_int)) {
    return (FALSE);
}
return (TRUE);
APPENDIX C

HCSdaemon

/*
 * This file assigns the initial benchmark data for all routines
 * known to libHCS.
 *
 * Since the routines are numbered sequentially from NULLPROC (0),
 * the routine names can be used as indices into the benchmark array.
 *
 * The fields of the benchmark structure are defined in libHCS.h and
 * are {routine, size, order, time} where
 *
 * routine - is the routine number
 * size - is the size (n) most recently benchmarked
 * order - is an enumerated value representing the order (big-O)
 * of calculation of the routine
 * time - is the benchmark execution time
 *
 * From the size, order and time information, one can estimate the
 * execution time of a given instance of the routine (see libHCS.c)
 */
int Global_Benchmark = 0;
int local_benchmark = 0;
int cnt = 7;
benchmark bench[MAXROUTINES] = {
    {0, 1, 0_1, -1.0},
    {ADDF, 1, 0_1, -1.0},
    {MULF, 1, 0_1, -1.0},
    {DIVF, 1, 0_1, -1.0},
    {DAXPY, 1, 0_N, -1.0},
    {DGESL, 1, 0_N2, -1.0},
    {DGEFA, 1, 0_N2, -1.0}
};
libHCS_svc.c

This is the main program for the server-side of the RPC protocol. This program receives requests for RPC service, and invokes the relevant procedure.

This source file was generated indirectly, by rpcgen(1) from the libHCS.x source file. Numerous modifications have been made to the original file as generated. These modifications are preceded by an 'HCSdaemon' comment block.

Note that rpcgen on the Cray, running UNICOS, generates a BSD 'bzero' call instead of the ISO 'memset'. Therefore, any occurrences of:

bzero((char ♦)&item, sizeof(item));

have been replaced with:

(void) memset((char ♦)&item, 0, sizeof (item));

Note: Adding an entry to libHCS_svc.c is one of the steps to adding a new library routine, func, to libHCS.

The following instructions describe the process of adding an entry to libHCS_svc.c:

1) Follow the directions found in libHCS.x for adding an entry to that protocol file.

2) Recompile the protocol file per the directions.

3) From the libHCS_svc.c file produced by step 2, extract the following and include at the corresponding locations in this file:

in hcsdaemon_l() -

in the argument union include the new entry
func_arg func_l_arg;

in the switch (rqstp->rq_proc) include the new entry

  case FUNC:
    xdr_argument = xdr_func_arg;
    xdr_result = xdr_func_res;
    local = (char *(*()) func_l;
    break;

4) For portability purposes, in the current implementation, replace the 'bzero' call as described above.
#include <stdio.h>
#include <rpc/rpc.h>
#include <signal.h>
#include <sys/times.h>
#include <time.h>
#include "libHCS.h"
#include "libBench.h"

static void hcsdaemon_l();

static struct timeval TIMEOUT = { 25, 0 };

main(int argc, char **argv)
{
    SVCXPRT *transp;

    if (!(argc == 1 || (argc == 2))) {
        fprintf(stderr,"%s: usage error, argv\n",argv[0]);
        exit(1);
    } else if (argc == 2)
    { if (strcmp(argv[1],"-b") != 0) {
            fprintf(stderr,"%s: usage error, argv[1]\n",argv[0]);
        exit(1);
    } else
        Global_Benchmark = 1;

    signal(SIGCHLD,SIG_IGN);
    (void)pmap_unset(HCSDAEMON, HCSVERS);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
        (void)fprintf(stderr, "cannot create udp service.\n");
        exit(1);
    }

    if (!svc_register(transp, HCSDAEMON, HCSVERS,
            hcsdaemon_l, IPPROTO_UDP)) {
        (void)fprintf(stderr, "unable to register (HCSDAEMON, HCSVERS, udp).\n");
        exit(1);
    }

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        (void)fprintf(stderr, "cannot create tcp service.\n");
    }
exit(1);
}
if (!svc_register(transp, HCSDAEMON, HCSVERS, hcsdaemon_l, IPPROTO_TCP)) {
  (void)fprintf(stderr, "unable to register (HCSDAEMON, HCSVERS, tcp).\n"));
  exit(1);
}

/*
 * HCSdaemon  Initialize internal tables - for now, just performance
 * HCSdaemon  values
 */
  init();

svc_run();
(void)fprintf(stderr, "svc_run returned\n");
exit(1);
}

static void 
hcsdaemon_l(rqstp, transp)
struct svc_req *rqstp;
SVCXPRT *transp;
{
  union {
    eight_k_arg eight_k_l_arg;
    addf_arg addf_l_arg;
    mulf_arg mulf_l_arg;
    divf_arg divf_l_arg;
    daxpy_arg daxpy_l_arg;
    dgesl_arg dgesl_l_arg;
    dgefa_arg dgefa_l_arg;
  } argument;
  char *result;
  bool_t (**xdr_argument)(), (**xdr_result());
  char **(*local)();

  switch (rqstp->rq_proc) {
    case NULLPROC:
      (void)svc_sendreply(transp, xdr_void, (char *)NULL);
      return;
    case EIGHT_K:
      xdr_argument = xdr_eight_k_arg;
      xdr_result = xdr_eight_k_res;
      local = (char **)(*local)() eight_k_l;
      break;
    case GETHOST:
      xdr_argument = xdr_void;
      xdr_result = xdr_gethost_res;
      local = (char **)(*local)() gethost_l;
      break;
  }
}
case ADDF:
    xdr_argument = xdr_addf_arg;
    xdr_result = xdr_float;
    local = (char (*)(())) addf_l;
    break;

case MULF:
    xdr_argument = xdr_mulf_arg;
    xdr_result = xdr_float;
    local = (char (*)(())) mulf_l;
    break;

case DIVF:
    xdr_argument = xdr_divf_arg;
    xdr_result = xdr_float;
    local = (char (*)(())) divf_l;
    break;

case DAXPY:
    xdr_argument = xdr_daxpy_arg;
    xdr_result = xdr_daxpy_res;
    local = (char (*)(())) daxpy_l;
    break;

case DGESL:
    xdr_argument = xdr_dgesl_arg;
    xdr_result = xdr_dgesl_res;
    local = (char (*)(())) dgesl_l;
    break;

case DGEFA:
    xdr_argument = xdr_dgefa_arg;
    xdr_result = xdr_dgefa_res;
    local = (char (*)(())) dgefa_l;
    break;

default:
    svcerr_noproc(transp);
    return;
}
(void) memset((char *)&argument, 0, sizeof (argument));
if (bench[rqstp->rq_proc].time == -1.0) local_benchmark = 1;

if (!svc_getargs(transp, xdr_argument, &argument)) {
    svcerr_decode(transp);
    return;
}

/*
 * HCSdaemon To allow for concurrent invocations of HCSdaemon,
 * HCSdaemon fork this copy off and go back to the main loop.
 */
if (fork() == 0) {
    result = (*local)(argument, rqstp);
    if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
        svcerr_sendreply(transp);
        return;
    }
}
svcerr_systemerr(transp);
}  
if (!svc_freeargs(transp, xdr_argument, &argument)) {
(void)fprintf(stderr, "unable to free arguments\n");
exit(1);
}
exit(0);

/*
 *HCSdaemon If we're benchmarking, wait for the child to finish
 *HCSdaemon updating the benchmark file.
 */
if (local_benchmark == 1) {
    wait();
    init();
}
local_benchmark = Global_Benchmark;

#endif CONVEX
/*
 * kludge for the lack of a properly operating
 * signal(SIGCHLD,SIG_IGN);
 * on the convex.
 */
wait();
#endif

/*
 *HCSdaemon Initialize the benchmarking system.
 */
init()
{
extern benchmark bench[];
extern int rcnt;
extern int Global_Benchmark;
extern int local_Benchmark;

FILE *fd;
long routine;
int size;
int order;
float time;

fd = fopen("libHCS.b","r");
if (fd)
    if (!Global_Benchmark) {
        rcnt = 0;
        while ((rcnt < MAXROUTINES) &&
                (fscanf(fd,"%ld %d %d %f",&routine,&size,&order,&time) !=
                 EOF)) {
            if ((routine < 0) || (routine > MAXROUTINES)) {
                fprintf(stderr,"init: internal error\n");
            }
        }
    }
}
exit(1);

else {
    bench[routine].routine = routine;
    bench[routine].size = size;
    bench[routine].order = order;
    bench[routine].time = time;
    rcnt++;
}

else
    if (fd) {
        fclose(fd);
        remove("libHCS.b");
    }

    Global_Benchmark = 0;
}
/*
 * libHCS_l.c
 * This is the source file where the actual subroutines are located.
 * These routines are invoked by calls from libHCS_svc.c after
 * suitable argument transfer and conversion (see libHCS_xdr.c).
 * Since the RPC clnt_call() routine has the means to pass a single
 * argument and return a single result, the original arguments and
 * results have been wrapped up into the structures func_arg and
 * func_res. func_arg must be unpacked, and its constituent parts
 * assigned to the elements of the original argument list (which has
 * been reproduced for this purpose.) Then, conversely, after the
 * routine has executed and results produced, these results must be
 * transferred from their local variables to the func_res structure
 * for return to the calling host. For this reason, a common return
 * point may have to be provided. If the routine is cleanly coded,
 * this should not present too much of a problem.
 * Bracketing the body of the subroutine are timing and benchmarking
 * calls. These are utilized whenever it is discovered that no
 * benchmarking data exists for the routine, or the Global_Benchmark
 * flag is set by virtue of invoking HCSdaemon with the -b (benchmark)
 * option.
 * In the following, comments which were added to the subroutines
 * being ported to libHCS appear as:

 *libHCS comment
 *libHCS more comments
 *
*/
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rstat.h>
#include <sys/times.h>
#include <time.h>
#include "libHCS.h"
extern benchmark bench[];
extern int rcnt;
extern int local_benchmark;
struct tms before, after;

func_res *func_l(struct func_arg *arg)
* Within the body of the function, the original arguments are
* reproduced as automatic variables. Also, a results structure
* is allocated.

    float a, b, c;
    static func_res res;

* Then, the next step is to assign to the local variables the
* corresponding fields from the argument structure so that the
* balance of the code will behave as if these values had been passed
* in normally via the argument list.

    a = arg->a;
    b = arg->b;
    c = arg->c;

* A timing call is made in case we are benchmarking.

    times(&before);

* The body of the original subroutine appears next.

    ...

* Followed by the balance of the benchmarking addition

    times(&after);
    if (local_benchmark) benchmark_l(FUNC,&before,&after,1);

* Finally, at a common return point (see some of the more complicated
* examples below for a better example) the return values are assigned
* to the relevant fields in the results structure before it is
* returned.

    res.d = d;
    return (&res);
}
 *
* And that’s it.
*/

/*
* addf_l, mulf_l and divf_l are test routines which were the first
* to be implemented in this project. They have been left intact
* because they provide simple, clear examples of the subroutine
* conversion process.
*/
addf_res *addf_l(struct addf_arg *arg)
{
    float x, y, z;
    static addf_res res;
x = arg->x;
y = arg->y;
times(&before);
    z = (x + y);
times(&after);
if (local_benchmark) benchmark_l(ADDF,&before,&after,1);
    res.z = z;
    return(&res);
}
mulf_res *mulf_l(struct mulf_arg *arg)
{
    float x,y,z;
    static mulf_res res;
    x = arg->x;
    y = arg->y;
times(&before);
    z = (x * y);
times(&after);
if (local_benchmark) benchmark_l(MULF,&before,&after,1);
    res.z = z;
    return(&res);
}
divf_res *divf_l(struct divf_arg *arg)
{
    float x,y,z;
    static divf_res res;
    x = arg->x;
    y = arg->y;
times(&before);
    z = (x / y);
times(&after);
if (local_benchmark) benchmark_l(DIVF,&before,&after,1);
    res.z = z;
    return(&res);
}
/*
 * The following routines have been converted from clinpack (the
* c version of the linpack benchmark.
*  
*  daxpy()
*  dgesl()
*  dgefa()
*  
* It should be noted that daxpy was the first subroutine that
* was attempted, but proved a poor choice since it has such low
* execution time. This, coupled with the fact that it is called
* over 100,000 times in the benchmark, makes it a bad choice for
* a distributed library.
*/

#include "clinpack.h"

/*
libHCS Hide the original function declaration and argument
libHCS specification
*/

/*
daxpy(n, da, dx, incx, dy, incy)
  
  constant times a vector plus a vector.
  jack dongarra, linpack, 3/11/78.

REAL dx[], dy[], da;
int incx, incy, n;
*/

/*
libHCS New function declaration to replace the original.
*/
daxpy_res *daxpy_l(daxpy_arg *arg)
{
/*
libHCS Declare the original argument list. This is followed
libHCS by any local declarations which appeared in the
libHCS subroutine as originally written.
*/
REAL *dx, *dy, da;
int incx, incy, n;

int i, ix, iy, m, mpl;

/*
libHCS Add a results structure declaration. Also, copy arguments
libHCS from the arg structure to the variables just declared. These
libHCS will be used in the program. Then set up benchmark timing.
*/
static daxpy_res res;

n = arg->n;
da = arg->da;
dx = arg->dx.dx_val;
in cx = arg->incx;
d y = arg->dy.dy_val;
incy = arg->incy;

times(&before);

/*
*libHCS Now, continue with the subroutine as originally written, with
*libHCS the exception that a common return point must be provided.
*libHCS If there are several return statements, a goto common_return
*libHCS must be inserted wherever a return is made.
*/
if (n <= 0) goto common_return;
if (da == ZERO) goto common_return;

if (incx != 1 || incy != 1) {
    /* code for unequal increments or equal increments
not equal to 1 */
    ix = 1;
iy = 1;
    if (incx < 0) ix = (-n+1)*incx + 1;
    if (incy < 0) iy = (-n+1)*incy + 1;
    for (i = 0; i < n; i++) {
        dy[iy] = dy[iy] + da*dx[ix];
        ix = ix + incx;
iy = iy + incy;
    }
    goto common_return;
}

/* code for both increments equal to 1 */
#ifdef ROLL
    for (i = 0; i < n; i++) {
        dy[i] = dy[i] + da*dx[i];
    }
#endif
#ifdef UNROLL
    m = n % 4;
    if (m != 0) {
        for (i = 0; i < m; i++)
            dy[i] = dy[i] + da*dx[i];
        if (n < 4) goto common_return;
    }
    for (i = m; i < n; i = i + 4) {
        dy[i] = dy[i] + da*dx[i];
        dy[i+1] = dy[i+1] + da*dx[i+1];
        dy[i+2] = dy[i+2] + da*dx[i+2];
dy[i+3] = dy[i+3] + da*dx[i+3];
    }
}
/*
 *libHCS common return point
 *libHCS Take care of benchmarking and then copy any return values
 *libHCS to the results structure before returning it to the calling
 *libHCS host.
 */

common_return:

 times(&after);

if (local_benchmark) benchmark_l(DIVF,&before,&after,1);

 let res.dy.dy_len = arg->dy.dy_len;
 let res.dy.dy_val = dy;

 return(&res);
}

/*
 *libHCS Hide the original function declaration and argument
 *libHCS specification
 */

/**------------------*/

/*
 dgesl (a, Ida, n, ipvt, b, job)

 int lda,n,ipvt[],job;
 REAL a[],b[];
 */

/*
 *libHCS New function declaration to replace the original.
 */

 dgesl_res *dgesl_l(dgesl_arg *arg)

/* We would like to declare a[][lda], but c does not allow it. In this
 function, references to a[i][j] are written a[lda*i+j]. */

/*
 dgesl solves the double precision system
 a * x = b or trans(a) * x = b
 using the factors computed by dgeco or dgefa.

 on entry

 a double precision[n][lda]
 the output from dgeco or dgefa.

 lda integer
 the leading dimension of the array a .

 n integer
 the order of the matrix a .

 ipvt integer[n]
 the pivot vector from dgeco or dgefa.
b double precision[n]
    the right hand side vector.

job integer
    = 0 to solve  a*x = b ,
    = nonzero to solve trans(a)*x = b where
    trans(a) is the transpose.

on return
b the solution vector x .

error condition

a division by zero will occur if the input factor contains a
zero on the diagonal. technically this indicates singularity
but it is often caused by improper arguments or improper
setting of lda . it will not occur if the subroutines are
called correctly and if dgeco has set rcond .gt. 0.0
or dgefa has set info .eq. 0 .

to compute inverse(a) * c where c is a matrix
with p columns
dgeco(a,lda,n,ipvt,rcond,z)
if (!rcond is too small){
    for (j=0,j<p,j++)
        dgesl(a,lda,n,ipvt,c[j][0],0);
}

linpack. this version dated 08/14/78.
cleve moler, university of new mexico, argonne national lab.

functions
blas daxpy,ddot
*/

{ /*
*libHCS Declare the original argument list. This is followed
*libHCS by any local declarations which appeared in the
*libHCS subroutine as originally written.
*/
    int lda,n,*ipvt,job;
    REAL *a,*b;

    REAL ddot(),t;
    int k,kb,l,nml;

/*
*libHCS Add a results structure declaration. Also, copy arguments
*libHCS from the arg structure to the variables just declared. These
*libHCS* will be used in the program. Then set up benchmark timing.

```c
static dgesl_res res;

a = arg->a.a_val;
lda = arg->lda;
n = arg->n;
ipvt = arg->ipvt.ipvt_val;
b = arg->b.b_val;
job = arg->job;
```

```c
times(&before);
```

*libHCS* Now, continue with the subroutine as originally written, with
*libHCS* the exception that a common return point must be provided.
*libHCS* If there are several return statements, a goto *common_return*
*libHCS* must be inserted wherever a return is made.

```c
nml = n - 1;
if (job == 0) {
    /* job = 0 , solve a * x = b
       first solve l*y = b */

    if (nml >= 1) {
        for (k = 0; k < nml; k++) {
            l = ipvt[k];
            t = b[l];
            if (l != k){
                b[l] = b[k];
                b[k] = t;
            }
            daxpy(n-(k+1),t,&a[lda*k+k+l], 1, &b[k+1],1);
        }
    }
    /* now solve u*x = y */

    for (kb = 0; kb < n; kb++) {
        k = n - (kb + 1);
        b[k] = b[k]/a[lda*k+k];
        t = -b[k];
        daxpy(k,t,&a[lda*k+0],1,&b[0], 1) ;
    }
} else {
    /* job = nonzero, solve trans(a) * x = b
       first solve trans(u)*y = b */

    for (k = 0; k < n; k++) {
        t = ddot(k,&a[lda*k+0],1,&b[0], 1);
        b[k] = (b[k] - t)/a[lda*k+k];
    }
}
```
/* now solve trans(1)*x = y */

if (nml >= 1) {
    for (kb = 1; kb < nml; kb++) {
        k = n - (kb+1);
        b[k] = b[k] + ddot(n-(k+1), &a[lda*k+k+1], 1, &b[k+1], 1);
        l = ipvt[k];
        if (l != k) {
            t = b[l];
            b[l] = b[k];
            b[k] = t;
        }
    }
}

/*
 *libHCS common return point
 *libHCS Take care of benchmarking and then copy any return values
 *libHCS to the results structure before returning it to the calling
 *libHCS host.
 */
times(&after);
if (local_benchmark) benchmark_l(DGESL,&before,&after,n);
    res.b.b_len = arg->b.b_len;
    res.b.b_val = b;
return(&res);
}

/*
 *libHCS Hide the original function declaration and argument
 *libHCS specification
 */
/**-----------------------------*/
/*
dgefa_l(dgefa_arg *arg)
REAL a[];
int lda,n,ipvt[],*info;
*/

/*
 *libHCS New function declaration to replace the original.
 */
dgefa_res *dgefa_l(dgefa_arg *arg)

/* We would like to declare a[][lda], but c does not allow it. In this
function, references to a[i][j] are written a[lda*i+j]. */

/*
dgefa factors a double precision matrix by gaussian elimination.
dgefa is usually called by dgeco, but it can be called
directly with a saving in time if rcond is not needed. 
(time for dgeco) = (1 + 9/n)* (time for dgefa)
.

on entry

a       REAL precision[n][lda]
the matrix to be factored.

lda    integer
the leading dimension of the array  a .

n     integer
the order of the matrix  a .

on return

a       an upper triangular matrix and the multipliers
which were used to obtain it.
the factorization can be written  a = l\cdot u  where
l  is a product of permutation and unit lower
triangular matrices and  u  is upper triangular.

ipvt     integer[n]
an integer vector of pivot indices.

info    integer
= 0 normal value.
= k     if \( u[k][k] .eq. 0.0 \) . this is not an error
condition for this subroutine, but it does
indicate that dgesl or dgedi will divide by zero
if called. use rcond in dgeco for a reliable
indication of singularity.

linpack. this version dated 08/14/78 .
cleve moler, university of new mexico, argonne national lab.

functions

 blas daxpy,dscal,idamax

*/

{ /*
 *libHCS Declare the original argument list. This is followed
 *libHCS by any local declarations which appeared in the
 *libHCS subroutine as originally written.
 */
REAL *a;
int lda,n,*ipvt,*info;

REAL t;
int idamax(),j,k,kpl,l,nml;
/*
*libHCS    Add a results structure declaration. Also, copy arguments
*libHCS    from the arg structure to the variables just declared. These
*libHCS    will be used in the program. Then set up benchmark timing.
*/
    static dgefa_res res;

    a = arg->a.a_val;
    lda = arg->lda;
    n = arg->n;
    ipvt = arg->ipvt.ipvt_val;
    info = arg->info;

    times(&before);
/*
*libHCS    Now, continue with the subroutine as originally written, with
*libHCS    the exception that a common return point must be provided.
*libHCS    If there are several return statements, a goto common_return
*libHCS    must be inserted wherever a return is made.
*/

    /* gaussian elimination with partial pivoting    */

    *info = 0;
    nml = n - 1;
    if (nml >= 0) {
        for (k = 0; k < nml; k++) {
            kpl = k + 1;

            /* find 1 = pivot index     */
            1 = idamax(n-k,&a[lda*k+k],1) + k;
            ipvt[k] = 1;

            /* zero pivot implies this column already
               triangularized */
            if (a[lda*k+l] != ZERO) {

                /* interchange if necessary */

                if (l != k) {
                    t = a[lda*k+l];
                    a[lda*k+l] = a[lda*k+k];
                    a[lda*k+k] = t;
                }

                /* compute multipliers */

                t = -ONE/a[lda*k+k];
                dscal(n-(k+1),t,&a[lda*k+k+l],1);
            }
        }
        /* row elimination with column indexing */
    }
for (j = kpl; j < n; j++) {
    t = a[lda*j+1];
    if (l != k) {
        a[lda*j+1] = a[lda*j+k];
        a[lda*j+k] = t;
    }
    daxpy(n-(k+1),t,&a[lda*k+k+l] , 1 , &a[lda*j+k+1], 1);
} else {
    *info = k;
}
}
ipvt[n-1] = n-1;
if (a[lda*(n-1)+(n-1)]  == ZERO) *info = n-1;

/*
 *libHCS common return point
 *libHCS Take care of benchmarking and then copy any return values
 *libHCS to the results structure before returning it to the calling
 *libHCS host.
 */
times(&after);
if (local_benchmark) benchmark_l(DGEFA,&before,&after,n);
    res.a.a_len = arg->a.a_len;
    res.a.a_val = a;

    res.ipvt.ipvt_len = n;
    res.ipvt.ipvt_val = ipvt;

    res.info = info;

    return(&res);
}

/*
 *libHCS The following are clinpack support routines for the networked
 *libHCS subroutines appearing above.
 */

/*-----------------------------*/
daxpy(n,da,dx,incx,dy,incy)
/*
 constant times a vector plus a vector.
 jack dongarra, linpack, 3/11/78.
 */
REAL dx[],dy[],da;
int incx,incy,n;
{ int i,ix,iy,m,ipl;
if(n <= 0) return;
if (da == ZERO) return;

if(incx != 1 || incy != 1) {
    /* code for unequal increments or equal increments
    not equal to 1 */
    ix = 1;
    iy = 1;
    if(incx < 0) ix = (-n+1)*incx + 1;
    if(incy < 0) iy = (-n+1)*incy + 1;
    for (i = 0; i < n; i++) {
        dy[iy] = dy[iy] + da*dx[ix];
        ix = ix + incx;
        iy = iy + incy;
    }
    return;
}

/* code for both increments equal to 1 */
#endif
#endif

m = n % 4;
if (m != 0) {
    for (i = 0; i < m; i++) {
        dy[i] = dy[i] + da*dx[i];
    }
} 
#endif

REAL ddot(n, dx, incx, dy, incy)
/*
 * forms the dot product of two vectors.
 * jack dongarra, linpack, 3/11/78.
 */
REAL dx[], dy[];

int incx, incy, n;
REAL dtemp;
int i,ix,iy,m,mpl;

dtemp = ZERO;

if(n <= 0) return(ZERO);

if(incx != 1 || incy != 1) {
    /* code for unequal increments or equal increments not equal to 1 */
    ix = 0;
    iy = 0;
    if (incx < 0) ix = (-n+1)*incx;
    if (incy < 0) iy = (-n+1)*incy;
    for (i = 0;i < n; i++) {
        dtemp = dtemp + dx[ix]*dy[iy];
        ix = ix + incx;
        iy = iy + incy;
    }
    return(dtemp);
}

/* code for both increments equal to 1 */

#ifdef ROLL
    for (i=0;i < n; i++)
        dtemp = dtemp + dx[i]*dy[i];
    return(dtemp);
#endif
#ifdef UNROLL
    #ifdef UNROLL
        m = n % 5;
        if (m != 0) {
            for (i = 0; i < m; i++)
                dtemp = dtemp + dx[i]*dy[i];
            if (n < 5) return(dtemp);
        }
        for (i = m; i < n; i = i + 5) {
            dtemp = dtemp + dx[i]*dy[i] +
                    dx[i+1]*dy[i+1] + dx[i+2]*dy[i+2] +
                    dx[i+3]*dy[i+3] + dx[i+4]*dy[i+4];
        }
        return(dtemp);
    #endif
#endif

/* scales a vector by a constant.
    jack dongarra, Linpack, 3/11/78. */
dscal(n,da,dx,incx)
REAL da, dx[];
int n, incx;
{
    int i, m, mpl, nincx;

    if(n <= 0) return;
    if(incx != 1) {
        /* code for increment not equal to 1 */
        nincx = n*incx;
        for (i = 0; i < nincx; i = i + incx)
            dx[i] = da*dx[i];
        return;
    }
    /* code for increment equal to 1 */

#ifdef ROLL
    for (i = 0; i < n; i++)
        dx[i] = da*dx[i];
#endif
#ifdef UNROLL
    m = n % 5;
    if (m != 0) {
        for (i = 0; i < m; i++)
            dx[i] = da*dx[i];
        if (n < 5) return;
    }
    for (i = m; i < n; i = i + 5) {
        dx[i] = da*dx[i];
        dx[i+1] = da*dx[i+1];
        dx[i+2] = da*dx[i+2];
        dx[i+3] = da*dx[i+3];
        dx[i+4] = da*dx[i+4];
    }
#endif

} /*---------------------------*/
int idamax(n, dx, incx)
/
    finds the index of element having max. absolute value.
    jack dongarra, linpack, 3/11/78.
*/

REAL dx[];
int incx, n;
{
    REAL dmax;
    int i, ix, itemp;
if( n < 1 ) return(-1);
if(n == 1 ) return(0);
if(incx != 1) {
    /* code for increment not equal to 1 */
    ix = 1;
    dmax = fabs((double)dx[0]);
    ix = ix + incx;
    for (i = 1; i < n; i++) {
        if(fabs((double)dx[ix]) > dmax) {
            itemp = i;
            dmax = fabs((double)dx[ix]);
        }
        ix = ix + incx;
    }
} else {
    /* code for increment equal to 1 */
    itemp = 0;
    dmax = fabs((double)dx[0]);
    for (i = 1; i < n; i++) {
        if(fabs((double)dx[i]) > dmax) {
            itemp = i;
            dmax = fabs((double)dx[i]);
        }
    }
    return (itemp);
}

/*
 * eight_k_l()
 * This is a simple buffer transfer routine which attempts
 * to provide an estimation of the network transfer time
 * required for argument and results transfers.
 */
eight_k_res *eight_k_l(eight_k_arg *arg)
{
    static eight_k_res res;
    int i;
    for (i = 0; i < 8192; i++)
        res.array[i] = arg->array[i];
    return(&res);
}

/*
 * gethost_l()
This routine returns system load and benchmark values for all
routines about which it knows.

The system load over the last minute (as returned by the rstat
library call) and adds 1 (to account for the client) is returned as
a percentage of available computing resource. For example, let’s
say that the current load returned by rstat is 3.2.
Then 3.2 + 1 = 4.2. This means that 4.2 processes will be sharing
the resource. So each will get

(1/4.2) = 24%

of the resource.

gethost_res *gethost_l(void *arg)
{
    static gethost_res res;
    struct statstime statp;
    int i;
    int stat;

    stat = rstat("localhost", &statp);

    if (stat == 0)
        res.load =
            100 * 1.0/ (((float) statp.avenrun[0]) / 256.0) + 1.0);
    else
        clnt_perrno(stat);

    res.rcnt = rcnt;
    for (i = 1; i < rcnt+1; i++)
        res.bench[i] = bench[i];

    return(&res);
}

benchmark_l(
    long routine, struct tms *before, struct tms *after, int size)
{
    clock_t ctime;
    float time;
    FILE *fd;
    int i;

    long lroutine;
    int lsize;
    int lorder;
float ltime;

c = (after->tms_utime - before->tms_utime)
    +
    (after->tms_stime - before->tms_stime);

time = (float) ctime/(float) CLK_TCK;

fd = fopen("libHCS.b","r");
i = 0;
if (fd)
    while ((i < MAXROUTINES) &&
        (fscanf(fd,"%ld %d %d %f",&lroutine,&lsize,&lorder,&ltime) !=
        EOF)) {
        if ((lroutine < 0) I (lroutine > MAXROUTINES)) {
            fprintf(stderr,"benchmark_l: internal error\n");
            exit(1);
        }
        else {
            bench[lroutine].routine = lroutine;
            bench[lroutine].size = lsize;
            bench[lroutine].order = lorder;
            bench[lroutine].time = ltime;
            i++;
        }
    }
if (rcnt == 0) rcnt = i;

fclose(fd);
if (routine < MAXROUTINES)
    if (bench[routine].time < time) {
        bench[routine].time = time;
        bench[routine].size = size;

        fd = fopen("libHCS.b","w");
        for (i = 0; i < rcnt; i++)
            fprintf(fd,"%ld %d %d %.6f\n",
                bench[i].routine,
                bench[i].size,
                bench[i].order,
                bench[i].time);
        fclose(fd);
    }
APPENDIX D

Simulator

/*
 * simu.h
 *
 * This is the header file for the libHCS simulator program.
 *
 */
#define MAXTRIALS 10
#define MAXHOSTS 4
#define MAXROUTINES 4
#define CLK_TCK 1.0
#define POLL_TIME 60
#define O_1 1
#define O_N 2
#define O_NLogN 3
#define O_N2 4
#define O_N3 5

float estimate_exec();
float estimate_xfer_latency();
float inverse_order();
float random_float();
float nlog2n_inv();
float nlog2n(float n);

struct benchmark {
    long routine;
    int size;
    int order;
    float time;
};

#define MAXHOSTLEN 2
struct host_characteristics {
    char *name[MAXHOSTLEN];
    int load, load_low, load_high, load_delta;
    float network_latency, netwl_low, netwl_high, netwl_delta;
    float xfer_latency, xferl_low, xferl_high, xferl_delta;
}
unsigned short *ld_rstream, *nl_rstream, *xl_rstream;
int rcnt;
struct benchmark bench[MAXROUTINES];
} hc_table[MAXHOSTS] = {
  "a",
  10, 3, 12, 1,
  .015, .010, .020, .001,
  .050, .040, .060, .005,
  0, 0, 0,
  4,
  0, 100, 2, .1,
  1, 150, 3, 2.5,
  2, 150, 4, 5.0,
  3, 60, 5, 20.0,
  "b",
  50, 30, 75, 5,
  .045, .030, .060, .003,
  .150, .120, .180, .015,
  0, 0, 0,
  4,
  0, 100, 2, 1.0,
  1, 150, 3, 10.0,
  2, 150, 4, 50.0,
  3, 60, 5, 200.0,
  "c",
  30, 20, 40, 2,
  .015, .010, .020, .001,
  .050, .040, .060, .005,
  0, 0, 0,
  4,
  0, 100, 2, .4,
  1, 150, 3, 10.0,
  2, 150, 4, 15.0,
  3, 60, 5, 80.0,
  "d",
  100, 95, 100, 1,
  .000, .000, .000, .001,
  .000, .000, .000, .005,
  0, 0, 0,
  4,
  0, 100, 2, 2.0,
  1, 150, 3, 50.0,
  2, 150, 4, 100.0,
  3, 60, 5, 400.0
};

struct host {
  char name[MAXHOSTLEN];
  int valid;
  int load;
  float network_latency;
  float xfer_latency;
  int rcnt;
  struct benchmark bench[MAXROUTINES];
float score[MAXROUTINES];
float exec_time[MAXROUTINES];

#define a 0
#define b 1
#define c 2
#define d 3

/*
#define CONFIG_MAX 4
  1, 0, 0, 0,
  1, 1, 0, 0,
  1, 0, 1, 0,
  1, 0, 0, 1
*/
#define MAXCONFIGS 15
int config_tbl[MAXCONFIGS][MAXHOSTS] = {
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 0, 1,
  1, 1, 0, 0,
  1, 0, 1, 0,
  1, 0, 0, 1,
  0, 1, 1, 0,
  0, 1, 0, 1,
  0, 0, 1, 1,
  1, 1, 1, 0,
  1, 1, 0, 1,
  1, 0, 1, 1,
  0, 1, 1, 1,
  1, 1, 1, 1
};

float Gtime_tally[MAXCONFIGS] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
/*
 *  simu.c
 *
 * This is the source code for a program to simulate the performance of
 * the libHCS/HCSdaemon distributed system.
 *
 * The library, libHCS, contains an opportunistic scheduling algorithm,
 * choose_host. This routine is called by each of the subroutines in
 * the library to select a remote host, running HCSdaemon, to which to
 * send the subroutine for execution (actually, the subroutine resides
 * in HCSdaemon on each of the candidate hosts, so only the argument
 * list is sent.) The algorithm seeks to minimize the execution time
 * of the subroutine by selecting the 'best' host for this purpose at
 * the time the subroutine call is made.
 *
 * Therefore, since the scheduling algorithm portion of this system is
 * wholly contained in the routine 'choose_host' in source file
 * libHCS.c, this is the routine, along with its support routines,
 * which this simulator drives.
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "simu.h"

/*
 * Host choice policies
 *
 * It is desirable to compare the choose_host() mechanism of host
 * selection to other selection policies. Those policies will
 * be: random choice, currently least loaded, and strongest (for
 * the current task.)
 */
#define CHOOSE_HOST 0
#define RANDOM 1
#define LEAST_LOADED 2
#define STRONGEST 3
int selection_policy;

/*
 * The overall results, by configuration, of the choose_host() policy
 * are sorted in descending elapsed time order. This sorting is used
 * for direct comparison of the choose_host() policy to the other
 * choice policies.
 */
int ch_cptr[MAXCONFIGS];

/*
 * Various global variables.
 */
float Gtime;
float watchdog_timer;
float last_hostupdate_time;
int tally[MAXCONFIGS][MAXHOSTS][MAXROUTINES];
int config;
int Gtally = 0;
int host_is_available[MAXHOSTS];

struct host hosts[MAXHOSTS];
int hostcnt = MAXHOSTS;
float trial_tally[MAXTRIALS][MAXCONFIGS];

int first_pass;

/*
 * Random number streams
 */
#define LD_RSTREAM_BASE 100001
unsigned short ld_rstream[MAXHOSTS][3];
#define NL_RSTREAM_BASE 200001
unsigned short nl_rstream[MAXHOSTS][3];
#define XL_RSTREAM_BASE 300001
unsigned short xl_rstream[MAXHOSTS][3];
#define HS_RSTREAM_BASE 400001
unsigned short hs_rstream[3];
unsigned short hs_rstream_save[3];

main()
{
    int i,j,k,l;
    int hst,rtn;

    for (selection_policy = CHOOSE_HOST; selection_policy < STRONGEST+1;)
        selection_policy++) {
        printf("selection policy \%d\n",selection_policy);
        for (config = 0; config < MAXCONFIGS; config++)
            for (i = 0; i < MAXHOSTS; i++)
                for (j = 0; j < MAXROUTINES; j++)
                    tally[config][i][j] = 0;

        for (hst = 0; hst < MAXHOSTS; hst++) {
            hc_table[hst].ld_rstream = &ld_rstream[hst][0];
            hc_table[hst].nl_rstream = &nl_rstream[hst][0];
            hc_table[hst].xl_rstream = &xl_rstream[hst][0];
        }

        for (i = 0; i < MAXCONFIGS; i++) Gtime_tally[i] = 0;

        for (k = 0; k < MAXTRIALS; k++) {
            for (i = 0; i < MAXCONFIGS; i++) trial_tally[k][i] = 0;

            if ((k % 1000) == 0) printf("%d'th trial\n",k);
        }
    /*
*/
Random number streams are maintained for the various portions of this simulation requiring random numbers. Also, these streams are saved across configuration changes so that each configuration can be tested under the same randomized conditions.

```c
for (hst = 0; hst < MAXHOSTS; hst++) {
  ld_rstream_save[hst][0] = 0; ld_rstream_save[hst][1] = 0;
  ld_rstream_save[hst][2] = 2 * (k*MAXHOSTS + hst) + LD_RSTREAM_BASE;
  nl_rstream_save[hst][0] = 0; nl_rstream_save[hst][1] = 0;
  nl_rstream_save[hst][2] = 2 * (k*MAXHOSTS + hst) + NL_RSTREAM_BASE;
  xl_rstream_save[hst][0] = 0; xl_rstream_save[hst][1] = 0;
  xl_rstream_save[hst][2] = 2 * (k*MAXHOSTS + hst) + XL_RSTREAM_BASE;
}
hs_rstream_save[0] = 0; hs_rstream_save[1] = 0;
hs_rstream_save[2] = k + HS_RSTREAM_BASE;
```

Execute this randomized trial in each configuration.

```c
for (config = 0; config < MAXCONFIGS; config++) {
  stream_copy(MAXHOSTS,ld_rstream,ld_rstream_save);
  stream_copy(MAXHOSTS,nl_rstream,nl_rstream_save);
  stream_copy(MAXHOSTS,xl_rstream,xl_rstream_save);
  /*
  * The starting parameters are the same for the execution of each configuration (see above.)
  */
  for (hst = 0; hst < MAXHOSTS; hst++) {
    host_is_available[hst] = config_tbl[config][hst];
    hc_table[hst].load = random_int(1,100,&ld_rstream[hst][0]);
    hc_table[hst].load_low = 1;
    hc_table[hst].load_high = 100;
    hc_table[hst].load_delta = 5;
    hc_table[hst].network_latency = random_float(.01,.10,&nl_rstream[hst][0]);
    hc_table[hst].netwl_low = .008;
    hc_table[hst].netwl_high = .016;
    hc_table[hst].netwl_delta = .0001;
    hc_table[hst].xfer_latency = random_float(.10,1.0,&xl_rstream[hst][0]);
    hc_table[hst].netwl_low = .008;
    hc_table[hst].netwl_high = .016;
    hc_table[hst].netwl_delta = .0002;
  }
  /*
/* Initialize the current run. */
Gtime = 0.0;
first_pass = 0;
watchdog_timer = 0.0;
last_hostupdate_time = 0.0;

/* Simulate the program run. */
for (i = 0; i < 20; i++) {
    subroutine(a,50,1000,1);
    subroutine(b,100,10000,1000);
    subroutine(c,100,10000,10);
    subroutine(d,50,10000,1);
}

/* Sum the elapsed time by configuration. */
for (i = 0; i < MAXCONFIGS; i++) print("%.1f","trial_tally[i]);
    printf("\n");
*/

print_results(selection_policy);

/* Print the results of all trials. */
for (config = 0; config < MAXCONFIGS; config++) {
    printf("configuration (%d,%d,%d,%d) ",
            config_tbl[config][0],
            config_tbl[config][1],
            config_tbl[config][2],
            config_tbl[config][3]);
    for (i = 0; i < MAXHOSTS; i++) {
        printf("tally[%2d][%2d] = %5d ",i,j,tally[config][i][j]);
        printf("\n");
    }
    printf("\n");
}

for (config = 0; config < MAXCONFIGS; config++) {
    printf("( %d,%d,%d) ",
            config_tbl[config][0],
            config_tbl[config][1],
            config_tbl[config][2],
            config_tbl[config][3]);
    printf("Gtime_tally[%2.2d] = %14.7f\n",Gtime_tally[config],Gtime_tally[config]);
}
config, Gtime_tally[config]/MAXTRIALS);
}
*/
}

/* subroutine() simulates the passage of time while the subroutine
* is being executed on the remote host.
*/

subroutine(sub_idx, size, sizeof_arg, sizeof_res)
  int sub_idx;
  int size;
  int sizeof_arg;
  int sizeof_res;
{
  float send_time;
  float exec_time;
  float recv_time;
  float rexec_time;
  int host_idx;

  int i, j;

  /* At this point in a real libHCS subroutine, we would be copying
   * argument lists (see any subroutine in libHCS.c.) This is
   * followed by a call to choose_host (Note that the simulatori
   * version of choose_host doesn't return the client pointer, but
   * the index of the host which won...
   */
  if (choose_host(sub_idx, size, sizeof_arg,
                sizeof_res, &host_idx) == 1) {
    printf("internal error\n");
    exit(1);
  }

  /* At this point in a real libHCS subroutine, we would call the
   * network version of our subroutine, on the host which was selected
   * by choose_host.
   * In the simulator we want to simply mark the passage of time while
   * the subroutine 'executes'. To accomplish this, we will need to
   * estimate the execution time on the remote host so that while the
   * remote subroutine is executing we can simulate the changing load
   * of the hosts in the network.
   * Also, we have to break out the send and receive times, because
   * the network load may have changed between the time the subroutine
   * was started and the time it ends...
   */
Take care of the send latency.

\[
\text{send\_time} = \text{estimate\_xfer\_latency} (\text{sizeof\_arg}, \text{&hosts}[\text{host\_idx}]); \\
\text{while} (\text{send\_time} > (\text{float})\text{POLL\_TIME}) \\
\quad \text{update\_sim} ((\text{float})\text{POLL\_TIME}); \\
\quad \text{send\_time} = \text{send\_time} - (\text{float})\text{POLL\_TIME}; \\
\text{if} (\text{send\_time} > 0.0) \\
\quad \text{update\_sim}(\text{send\_time});
\]

Take care of subroutine execution time. This is complicated by
the fact that as the load on the remote host changes, so does the
execution time of the remaining portion of the algorithm.
Therefore, at each passage of POLL\_TIME time we need to re-
estimate the remaining execution time. Since the original
estimation operation requires the application of the time
complexity function to the current problem size, the inverse
operation requires the application of the inverse of the
* respective time complexity function. See 'inverse\_order()'.

\[
\text{exec\_time} = \text{estimate\_exec} (\text{sub\_idx}, \text{&hosts}[\text{host\_idx}], \text{size}); \\
\text{while} (\text{exec\_time} > (\text{float})\text{POLL\_TIME}) \\
\quad \text{update\_sim} ((\text{float})\text{POLL\_TIME}); \\
\quad \text{rexec\_time} = \text{exec\_time} - (\text{float})\text{POLL\_TIME}; \\
\quad \text{rtime\_ratio} = \text{rexec\_time}/\text{exec\_time}; \\
\quad \text{size} = \text{inverse\_order} (\text{host\_idx}, \text{sub\_idx}, \text{rexec\_time}, \text{exec\_time})*\text{size}; \\
\quad \text{exec\_time} = \text{estimate\_exec} (\text{sub\_idx}, \text{&hosts}[\text{host\_idx}], \text{size}); \\
\text{if} (\text{exec\_time} > 0.0) \\
\quad \text{update\_sim}(\text{exec\_time});
\]

Take care of receive latency.

\[
\text{recv\_time} = \text{estimate\_xfer\_latency} (\text{sizeof\_res}, \text{&hosts}[\text{host\_idx}]); \\
\text{while} (\text{recv\_time} > (\text{float})\text{POLL\_TIME}) \\
\quad \text{update\_sim} ((\text{float})\text{POLL\_TIME}); \\
\quad \text{recv\_time} = \text{recv\_time} - (\text{float})\text{POLL\_TIME}; \\
\text{if} (\text{recv\_time} > 0.0) \\
\quad \text{update\_sim}(\text{recv\_time}); \\
\text{return;}
\]

There are a few differences between the production choose\_host and
our simulator version. First of all, several variables have been
moved to Global scope:

- o hosts function 'subroutine()' needs the host
information

* o first_pass
  there are as many 'first_pass' es as there are
* simulated executions (see main())
* o watchdog_timer
* o hostcnt
* Also, several variables have been removed altogether:
* * o dummy
* * o gethost_arg
* * hostlist
* * o eight_k_arg
* * eight_k_res
*/

int choose_host(int routine, int size, int sizeof_arg,
  int sizeof_res, int *host_idx)
{
  float temp_watchdog;
  float xfer_ratio;
  float send_latency;
  float recv_latency;

  int i, j;

  if (first_pass == 0) {
    first_pass = 1;
    /*
    * In the production code, the call to gethostlist is here.
    * In the simulator, we have a fixed array of hosts.
    * The only thing left is the name initialization and setting
    * the valid flag to 0.
    *
    * Also, we need a means of inactivating non-participating
    * hosts. The host_is_available array is used for this purpose,
    * and will be seen later on.
    */
    for (i = 0, j = 0; i < hostcnt; i++, j++) {
      strcpy(hosts[j].name, hc_table[i].name);
      hosts[j].valid = 0;
    }
  }
  /*
  * check our watchdog timer to see if it's time to re-poll
  * In the simulator, the call to times has been replaced with Gtime
  * (expressed in seconds - so the CLK_TCK factor set to 1...)
  */
  if (((temp_watchdog = Gtime) - watchdog_timer)/(float)CLK_TCK) >
    30.0) {
    int j;
watchdog_timer = temp_watchdog;
for (j = 0; j < hostcnt; j++)
    hosts[j].valid = 0;
}

/*
 *  Now go through the list and discover the best candidate for this
 *  task.  This process makes the following assumptions:
 *  *
 *  o Each host reports its load as a percentage of 'available'
 *     processing potential (including the current function
 *     execution, if selected.)
 *  *
 *  o Network latency can be discovered with a simple NULLPROC
 *     RPC call.
 *  *
 *  o Transfer latency can be discovered by transferring an 8KB
 *     buffer.  This latency value can then be used to estimate
 *     the time required to send arguments and return results.
 *  *
 *  o Benchmark data is present, per host, for all callable
 *     routines, which includes size and time for the benchmark
 *     execution as well as the order of execution complexity.
 *     This information can be used to estimate the execution
 *     time of an instance of the routine with size
 *  *
 *  In the simulator, no actual network calls are necessary.  values
 *  are used from the host_characteristics array (hc_table).
 */
for (i = 0; i < hostcnt; i++)
{
    /*
     *  If we haven't already gotten load and performance information
     *  from server i, do so now.
     *
     *  In the simulator, the only thing left in this segment is to
     *  copy values from hc_table.  Also, note the added
     *  host_is_available condition to achieve a variable network
     *  configuration.
     */
    if ((hosts[i].valid == 0) && (host_is_available[i] == 1)) {
        /*
         *  Make sure we can get to this host and, as a side effect,
         *  discover its network latency (via a NULLPROC call).
         */
        hosts[i].network_latency = hc_table[i].network_latency;

        /*
         *  next, discover its 8K transfer latency
         */
        hosts[i].xfer_latency = hc_table[i].xfer_latency;

        /*
         *  Finally, get this host's load and the benchmark data for
         *  all routines known to it.  Then invalidate this host's
         */
    }
* scores for all routines.
*/
hosts[i].load = hc_table[i].load;
hosts[i].rcnt = hc_table[i].rcnt;
for (j = 0; j < hosts[i].rcnt; j++) {
    hosts[i].bench[j] = hc_table[i].bench[j];
    hosts[i].score[j] = -1;
}
hosts[i].valid = host_is_available[i];
}

/*
* OK, calculate, if necessary, each server's score for this routine
*/
for (i = 0; i < hostcnt; i++)
    if (hosts[i].valid == 1)
        if (routine >= hosts[i].rcnt) {
            fprintf(stderr,"internal error\n");
            fprintf(stderr,"marking %s as invalid\n",hosts[i].name);
            hosts[i].valid = 0;
        } else
            /*
             * The score calculation is the sum of the send latency,
             * execution time, and receive latency.
            */
            if (hosts[i].score[routine] == -1) {
                send_latency =
                    estimate_xfer_latency(sizeof_arg, &hosts[i]);
                hosts[i].exec_time[routine] =
                    estimate_exec(routine, &hosts[i], size);
                recv_latency =
                    estimate_xfer_latency(sizeof_res, &hosts[i]);
                hosts[i].score[routine] =
                    send_latency +
                    hosts[i].exec_time[routine] +
                    recv_latency;
            }

switch (selection_policy) {
    case CHOOSE_HOST:
        /*
         * Pick the best score (lowest time) from all hosts.
         *
         * In the simulator version we return the host index instead
         * of the client handle. Also, we do some rudimentary
         * statistics gathering here.
        */
        *host_idx = hselect(hostcnt, hosts, routine);
        break;
    case RANDOM:
        /*
         * or, pick a host at random
*host_idx = rselect(hostcnt, hosts);
break;
case LEAST_LOADED:
  /*
   *  or, pick the least loaded host
   *
   */
  *host_idx = lselect(hostcnt, hosts);
break;
case STRONGEST:
  /*
   *  or, pick the 'strongest' host
   */
  *host_idx = sselect(hostcnt, hosts, routine);
break;
}
tally[config][*host_idx][routine] += 1;
Gtally += 1;
return(0);

int hselect(hostcnt, hosts, routine)
int hostcnt;
struct host hosts[];
long routine;
{
  int computed_index[10];
  int i, minindex;
  float min;

  min = 99999999.9;
  minindex = 0;
  for (i = 0; i < hostcnt; i++)
    if (hosts[i].valid && (hosts[i].score[routine] < min))
      { min = hosts[i].score[routine];
        minindex = i;
      }
return(minindex);
}

int rselect(hostcnt, hosts)
int hostcnt;
struct host hosts[];
{
int cnt = 0;
int rnd;
int index_list[MAXHOSTS];
int i;

for (i = 0; i < hostcnt; i++)
    if (hosts[i].valid)
        index_list[cnt++] = i;

rnd = random_int(0,cnt-1, hs_rstream);
return(index_list[rnd]);

/*
 * pick the least loaded host
*/
int lselect(hostcnt, hosts)
int hostcnt;
struct host hosts[];
{
    int minload = 0;
    int i, minindex;

    for (i = 0; i < hostcnt; i++)
        if ((hosts[i].valid) && (hosts[i].load > minload)) {
            minload = hosts[i].load;
            minindex = i;
        }

    return(minindex);
}

/*
 * pick the strongest host for this routine
*/
int sselect(hostcnt, hosts, routine)
int hostcnt;
struct host hosts[];
long routine;
{
    int i, minindex;
    float time;

    time = 9999999.0;
    minindex = 0;

    for (i = 0; i < hostcnt; i++)
        if (hosts[i].valid && (hosts[i].bench[routine].time < time)) {
            time = hosts[i].bench[routine].time;
            minindex = i;
        }
    return(minindex);
}
float estimate_xfer_latency(xfer_size, hostp)
    int xfer_size;
    struct host *hostp;
{
    float xfer_ratio;
    float latency;

    xfer_ratio = ((float) (xfer_size))/8192.0;
    latency = hostp->network_latency;
    latency+= (hostp->network_latency < hostp->xfer_latency)?
        xfer_ratio * (hostp->xfer_latency - hostp->network_latency) :
        xfer_ratio * hostp->xfer_latency;

    return (latency);
}

float estimate_exec(routine, hostp, size)
    int routine;
    struct host *hostp;
    int size;
{
    float size_ratio;
    float time_ratio;
    float exec_time;

    /*
     * assume valid numbers
     */
    size_ratio = (float)size/(float)hostp->bench[routine].size;

    switch (hostp->bench[routine].order) {
        case O_1:
            time_ratio = 1:
            break;
        /*
         * linear relationship => n'/n
         */
        case O_N:
            time_ratio = size_ratio;
            break;
        /*
         * nlogn case => n'*log(n')/n*log(n) =>
         * size_ratio * log(n')/log(n)
         */
        case O_NLogN:
            time_ratio =
            size_ratio * log(size)/log(hostp->bench[routine].size);
            break;
        /*
         * n^2 case => n'^2/n^2 => (n' * n')/(n * n) =>
         * size_ratio * size_ratio
         * (same for n^3)
         */
        case O_N2:


time_ratio = size_ratio * size_ratio;
break;
case O_N3:
    time_ratio = size_ratio * size_ratio * size_ratio;
    break;
}

exec_time = (time_ratio * hostp->bench[routine].time)/
            (((float)hostp->load)/100.0);

return(exec_time);
}

/*
 * update_sim()
 *
 * Account for the passage of time in the simulation.
 */

update_sim(time)
    float time;
{
    int i;

    Gtime+= time;

    if ((Gtime - last_hostupdate_time) >= (float)POLL_TIME) {
        last_hostupdate_time = Gtime;
        for (i = 0; i < hostcnt; i++)
            update_host(&hc_table[i]);
    }
}

/*
 * update_host()
 *
 * Simulate a changing environment.
 */

update_host(hostp)
    struct host_characteristics *hostp;
{
    int new_load;
    float new_network_latency;
    float new_xfer_latency;

    /*
     * Calculate a new load for this host and insure it is within the
     * global constraints.
     */
    new_load = hostp->load +
               random_int(-(hostp->load_delta), hostp->load_delta,
                            hostp->ld_rstream);

    if (new_load < hostp->load_low)
        hostp->load = hostp->load_low;
    else if (new_load > hostp->load_high)
hostp->load = hostp->load_high;
else
    hostp->load = new_load;

/*
 * Calculate a new network latency to this host and insure it is
 * within the global constraints.
 */
new_network_latency = hostp->network_latency +
    random_float(-(hostp->netwl_delta), hostp->netwl_delta,
        hostp->nl_rstream);

if (new_network_latency < hostp->netwl_low)
    hostp->network_latency = hostp->netwl_low;
else if (new_network_latency > hostp->netwl_high)
    hostp->network_latency = hostp->netwl_high;
else
    hostp->network_latency = new_network_latency;

/*
 * Calculate a new transfer latency to this host and insure it is
 * within the global constraints.
 */
new_xfer_latency = hostp->xfer_latency +
    random_float(-(hostp->xferl_delta), hostp->xferl_delta,
        hostp->xl_rstream);

if (new_xfer_latency < hostp->xferl_low)
    hostp->xfer_latency = hostp->xferl_low;
else if (new_xfer_latency > hostp->xferl_high)
    hostp->xfer_latency = hostp->xferl_high;
else
    hostp->xfer_latency = new_xfer_latency;

/*====================================================================*/

float random_float(float low, float high, unsigned short stream[3])
{
    return ((float)erand48(stream) * (high-low)+low);
}
*/
* random_int()
*
* Return a uniformly distributed, random integer within the
* interval
*  
* [low,high]
* 
* Note the use of the pre-initialized stream variable (see
* man(1) erand48)
*/
int random_int(int low, int high, unsigned short stream)
{
int low;
int high;
unsigned short stream[3];
{
return ((int)(erand48(stream)* (high-low+1))+low);
}

*/
* inverse_order()
*
* This routine allows us to deal with a partially executed task
* which has been 'interrupted' (in our simulation) to update the
* global state of the distributed computer. That is, if the current
* task runs long enough, the load on the computer executing it will
* change. We must be able to account for that change. By using the
* inverse_order() call, we can discover how much of the original
* task has been completed, and then, after updating the load,
* continue simulation on the balance of the task.
* 
* Note that time_ratio alone, is insufficient because, in general
* 
* -1
*   
* f (t')/f (t) is not equal to f (t'/t)
* 
*/
float inverse_order(int host_idx, int sub_idx, float tprime, float t)
{
float inverse;
float time_ratio;

time_ratio = tprime/t;

switch (hosts[host_idx].bench[sub_idx].order) {
    case 0_1:
        inverse = 1.0;
        break;
    case 0_N:
        inverse = time_ratio;
break;
case 0_NLogN:
    inverse = nlog2n_inv(tprime)/nlog2n_inv(t);
    break;

    /*
    * n^2 case => t'^(1/2)/t^(1/2) =>
    * (t'/t)^(1/2) => time_ratio^(1/2)
    * (same for n^3)
    */
case 0_N2:
    inverse = (float)pow((double)time_ratio, (double) (1.0/2.0));
    break;
case 0_N3:
    inverse = (float)pow((double)time_ratio, (double) (1.0/3.0));
    break;
}
return (inverse);

float nlog2n_inv(C)
    float C;
{
    float delta = .00001;
    float l2 = log((double)2.0);
    float f; 
    float df;
    float xnew;

    xnew = 1;
    while (nlog2n(xnew) < C)
        xnew*= 2;

    while ((nlog2n(xnew) - C) > delta) {
        f = nlog2n(xnew);
        df = (1.0 + log(xnew))/l2;
        xnew = xnew - (f - C)/df;
    }
    return(xnew);
}

float nlog2n()
float nlog2n(float n)
{
    return(n * (float)log((double)n)/log((double)2.0));
}

stream_copy(int size, unsigned short *pl, unsigned short *p2)
{
    int i;
    for (i = 0; i < size * 3; i++)
        pl[i] = p2[i];
}

print_results(int selection_policy)
{
    float trialmax;
    int cptr[MAXCONFIGS];
    int tmp;
    int i, j, config;
    FILE *plt, *ld, *key, *ooo;
    FILE *ch_plt, *ch_ld, *ch_key;
    struct ytic {
        char label[10];
        int value;
    } ytics[100];
    int ytic_cnt;
    int ooocount;
    char dir[30];
    char *policy_dir[STRONGEST+1] =
    {"choose_host","random","least_loaded","strongest"};
    for (config = 0; config < MAXCONFIGS; config++)
        cptr[config] = config;

    for (i = 0; i < MAXCONFIGS - 1; i++)
        for (j = 0; j < MAXCONFIGS - (i+1); j++)
if (Gtime_tally[cptr[j]] < Gtime_tally[cptr[j+1]]) {
  tmp = cptr[j];
  cptr[j] = cptr[j+1];
  cptr[j+1] = tmp;
}

/*
 * Save the configuration order of the choose_host trial so that
 * we can compare to other policies.
 */
if (selection_policy == CHOOSE_HOST) {
  for (i = 0; i < MAXCONFIGS; i++) ch_cptr[i] = cptr[i];

  /*
   * For the choose_host() policy attempt to discover, for
   * each trial, the correlation between the configuration
   * ordering of the individual trial and the overall average
   * of all trials.
   */
  ooo = fopen("out_of_order","w");
  for (i = 0; i < MAXTRIALS; i++) {
    trialmax = trial_tally[i][ch_cptr[0]];
    ooocount = 0;
    for (config = 0; config < MAXCONFIGS; config++) {
      fprintf(ooo,"%d ", (int)trial_tally[i][ch_cptr[config]]);
      if (trialmax < trial_tally[i][ch_cptr[config]]) {
        fprintf(ooo,"<%d> ", config+1);
        ooocount+= 1;
      }
      trialmax = trial_tally[i][ch_cptr[config]];
    }
    fprintf(ooo,"\n");
    fprintf(ooo,"trial %d is out of order %d times\n",i, ooocount);
  }
}

/*
 * Set up the ytic (for gnuplot) values and strings.
 */
for (i = 1; i < 21; i++) {
  strcpy(ytics[i].label," ");
  ytics[i].value = i * 1000;
}
strcpy(ytics[ 1].label, "1000");
strcpy(ytics[ 2].label, "2000");
strcpy(ytics[ 3].label, "3000");
strcpy(ytics[ 4].label, "4000");
strcpy(ytics[ 5].label, "5000");
strcpy(ytics[10].label,"10000");
strcpy(ytics[15].label,"15000");
strcpy(ytics[20].label,"20000");

ckdir("graphs");
ckdir(policy_dir[selection_policy]);
pl = fopen("results.plt","w");
l = fopen("results.ld", "w");
key = fopen("results.key" , "w");
ch_pl = fopen("ch_results.plt", "w");
l = fopen("ch_results.ld", "w");
ch_key = fopen("ch_results.key", "w");

/*
 * For each of the file types (pl, l, key) save first by the
 * sort order for the current policy, followed by a save with
 * the sort order for the choose_host() policy.
 */
for (config = 0; config < MAXCONFIGS; config++) {
        fprintf(pl, "%d %14.7f\n", config+1, Gtime_tally[cptr[config]]/MAXTRIALS);
        fprintf(ch_pl, "%d %14.7f\n", config+1, Gtime_tally[ch_cptr[config]]/MAXTRIALS);
}

for (config = 0; config < MAXCONFIGS; config++) {
        fprintf(key, "%d (%d,%d,%d,%d) %5d\n",config+1,
            config_tbl[cptr[config]][0],
            config_tbl[cptr[config]][1],
            config_tbl[cptr[config]][2],
            config_tbl[cptr[config]][3],
            (int) Gtime_tally[cptr[config]]/MAXTRIALS);
        fprintf(ch_key, "%d (%d,%d,%d,%d) %5d\n",config+1,
            config_tbl[ch_cptr[config]][0],
            config_tbl[ch_cptr[config]][1],
            config_tbl[ch_cptr[config]][2],
            config_tbl[ch_cptr[config]][3],
            (int) Gtime_tally[ch_cptr[config]]/MAXTRIALS);
}

fprintf(l, "set term postscript eps \"Times-Roman\" 22\n");
fprintf(l, "#set term mif\n");
fprintf(l, "set output \"results.ps\"\n");
fprintf(l, "#set output \"results.mif\"\n");
fprintf(l, "set logscale y\n");
fprintf(l, "set nokey\n");
fprintf(l, "set ylabel \"Total seconds\" 2,0\n");
fprintf(l, "set xlabel \"configuration (see accompanying key)\" 0,-1\n");
fprintf(l, "set xtics ("1" 1, "2" 2, "3" 3, "4" 4, "5" 5, "6" 6, "7" 7, "8" 8, "9" 9, "10" 10, "11" 11, "12" 12, "13" 13, "14" 14, "15" 15)\n");

for (i = 1; i < 21; i++) {
        strcpy(ytics[i].label, "");
        ytics[i].value = i * 1000;
}
strcpy(ytics[ 1].label, "1000");
strcpy(ytics[ 2].label, "2000");
strcpy(ytics[3].label, "3000");
strcpy(ytics[4].label, "4000");
strcpy(ytics[5].label, "5000");
strcpy(ytics[10].label, "10000");
strcpy(ytics[15].label, "15000");
strcpy(ytics[20].label, "20000");

fprintf(Id, "set ytics (\\n");
for (i = 1; i < 20; i++)
    fprintf(Id, "\"%s\" %d, \n", ytics[i].label, ytics[i].value);
fprintf(Id, "\"%s\" %d \n", ytics[20].label, ytics[20].value);
fprintf(Id, ")\\n");

fprintf(Id,
    "plot [0:16][500:20000] \"results.plt\" with linespoints\\n");

fprintf(ch_ld, "set term postscript eps \"Times-Roman\" 22\\n");
fprintf(ch_ld, "set term mif\\n");
fprintf(ch_ld, "set output \"ch_results.ps\\n");
fprintf(ch_ld, "set logscale y\\n");
fprintf(ch_ld, "set nokey\\n");
fprintf(ch_ld, "set ytics (\\n");
for (i = 1; i < 21; i++) {
    strcpy(ytics[i].label, " ");
    ytics[i].value = i * 1000;
}
strcpy(ytics[1].label, "1000");
strcpy(ytics[2].label, "2000");
strcpy(ytics[3].label, "3000");
strcpy(ytics[4].label, "4000");
strcpy(ytics[5].label, "5000");
strcpy(ytics[10].label, "10000");
strcpy(ytics[15].label, "15000");
strcpy(ytics[20].label, "20000");

fprintf(ch_ld, "set ytics (\\n")
for (i = 1; i < 20; i++)
    fprintf(ch_ld, "\"%s\" %d, \n", ytics[i].label, ytics[i].value);
fprintf(ch_ld, "\"%s\" %d \n", ytics[20].label, ytics[20].value);
fprintf(ch_ld, ")\\n");

fprintf(ch_ld,
    "plot [0:16][500:20000] \"results.plt\" with linespoints\\n");

close(plt);
close(Id);
close(key);
close(ch_plt);
close(ch_ld);
close(ch_key);

chdir("../../");
BIBLIOGRAPHY


