

Getting Started With Array Systems

Document Number 007-3058-003

CONTRIBUTORS

Updated by David Cortesi from earlier editions by David Cortesi and Carolyn Curtis
Illustrated by Dany Galgani and Brian Totty
Production by Cindy Stief
Engineering contributions by (in alphabetical order) Rob Bradshaw, John Brown, Ajit
Dandapani, Ira Pramanick, Lynd Stringer, Brian Totty, and Gary Walters
St Peter's Basilica image courtesy of ENEL SpA and InfoByte SpA. Disk Thrower
image courtesy of Xavier Berenguer, Animatica.

© 1996 - 1997, Silicon Graphics, Inc.— All Rights Reserved

The contents of this document may not be copied or duplicated in any form, in whole
or in part, without the prior written permission of Silicon Graphics, Inc.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure of the technical data contained in this document by
the Government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the
Rights in Technical Data and Computer Software clause at DFARS 52.227-7013
and/or in similar or successor clauses in the FAR, or in the DOD or NASA FAR
Supplement. Unpublished rights reserved under the Copyright Laws of the United
States. Contractor / manufacturer is Silicon Graphics, Inc., 2011 N. Shoreline Blvd.,
Mountain View, CA 94043-1389.

Silicon Graphics, CHALLENGE, and IRIS are registered trademarks and
CHALLENGEcomplib, Developer Magic, Extreme Graphics, Indigo Magic, Indy,
InSight, IRISconsole, IRISconsole ST-1600,, ST-1616, IRIX, IRIXPro, NetVisualyzer,
Origin2000, Origin200, Onyx2, OCTANE, O2, Performance Co-Pilot, POWER
CHALLENGE, POWER CHALLENGEarray, POWER CHALLENGE R10000,
POWERnode, POWER Onyx, POWERpath-2, ProDev, RealityEngine², and XFS are
trademarks of Silicon Graphics, Inc.

MIPSpro is a trademark of MIPS Technologies, Inc.
UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.
NFS is a trademark of Sun Microsystems, Inc.
ES-16 is a trademark of Essential Communications, Inc.
PerfAcct and Instrumental are trademarks of Instrumental, Inc.
Applied Parallel Research is a trademark of Applied Parallel Research, Inc.
Platform Computing and LSF are trademarks of Platform Computing Corporation.
SHARE II, Fair Share, and Hibernator II are trademarks of Softway Pty Ltd.
The Portland Group is a trademark of the Portland Group, Inc.
Applied Parallel Research and APR are trademarks of Applied Parallel Research, Inc.
Codine is a trademark of Genias Software, Inc.

Contents

List of Figures ix

List of Tables xi

About This Guide xiii

Audience xiii

Structure of This Guide xiv

Conventions Used in This Guide xv

- 1. Array System Components** 1
 - Array Components 2
 - Array Hardware Components 2
 - Array Software Components 3
 - Array Architecture 5
 - Array Nodes 5
 - The HIPPI Interconnect 6
 - Visualization and Interactive Supercomputing 7
 - Centralized Console Management 9
 - Distributed Management Tools 10
 - Array Services 10
 - Performance Co-Pilot 11
 - SHARE II (Fair Share) Scheduling 12
 - Accounting With PerfAcct 12
 - Supporting Documentation 13
 - Job Execution Facilities 13
 - Interactive Processing 13
 - Batch Processing 14

- Compilation, Development, and Execution Facilities 16
 - Optimizing and Parallelizing Compilers 17
 - High Performance Fortran 18
 - Numerical Libraries 18
 - IRIX 6.2 and 6.4 19
 - Performance and Debugging Tools 20
- Message-Passing Protocols 21
- 2. Using an Array 23**
 - Using an Array System 24
 - Finding Basic Usage Information 24
 - Logging In to an Array 25
 - Invoking a Program 25
 - Managing Local Processes 26
 - Monitoring Processes and System Usage 26
 - Scheduling and Killing Local Processes 27
 - Summary of Process Management Commands 27
 - Managing Batch Jobs with NQE 28
 - Accessing the NQE Commands 28
 - Starting NQE 28
 - Checking Job Status with NQE 29
 - Submitting a Job with NQE 30
 - About NQE Command Line Interfaces 32
 - Using Array Services Commands 33
 - About Array Sessions 33
 - About Names of Arrays and Nodes 34
 - About Authentication Keys 34
 - Summary of Common Command Options 34
 - Common Environment Variables 36

Interrogating the Array	37
Learning Array Names	37
Learning Node Names	37
Learning Node Features	38
Learning User Names and Workload	38
Browsing With ArrayView	40
Managing Distributed Processes	41
About Array Session Handles (ASH)	41
Listing Processes and ASH Values	42
Controlling Processes	42
3. Administering an Array	47
Using the IRISconsole Workstation	48
Using Other Tools With IRISconsole	52
About Array Configuration	53
About the Uses of the Configuration File	53
About Configuration File Format and Contents	54
Loading Configuration Data	54
About Substitution Syntax	55
Testing Configuration Changes	56
Configuring Arrays and Machines	57
Specifying Arrayname and Machine Names	57
Specifying IP Addresses and Ports	57
Specifying Additional Attributes	58
Configuring Authentication Codes	58
Configuring Array Commands	58
Operation of Array Commands	59
Summary of Command Definition Syntax	59
Configuring Local Options	62
Designing New Array Commands	62

- 4. Performance-Driven Programming in Array 3.0 65**
 - Basic Array Application Tuning Strategy 66
 - Tuning Single-Node Performance 66
 - Parallel Performance Goals 69
 - Designing Appropriate Parallel Algorithms 69
 - Test and Debug on a Single-node Server 70
 - Parallel Programming and Communication Paradigms 70
 - Shared-Memory Communication 71
 - Message-Passing IPC 71
 - Hybrid Models 72
 - Locality, Latency, and Bandwidth 73
 - MPI Communication Delays 74
 - TCP/IP Communication Delays 74
 - Reducing the Effect of Communication Delay 74
 - Array Services Library 76
 - Array Services Library Overview 76
 - Connecting to Array Services Daemons 78
 - Database Interrogation 80
 - Managing Array Service Handles 80
 - Executing an array Command 81
 - Executing a User Command 84
- A. The RendAsunder Demo Program 85**
 - Starting RendAsunder 86
 - Setting Up the Configuration File 86
 - Setting the Graphics Display 86
 - The Graphics Window 87
 - The Controls Window 88
 - Controls Window Menus 88
 - Controls Window Sliders 89
 - For More Information 90
- B. Array Documentation Quick Reference 91**

List of Figures

Figure 1-1	Array System Schematic	3
Figure 1-2	Advanced Visualization With Arrays	8
Figure 1-3	Batch Processing on an Array System	15
Figure 2-1	NQE Top-level Window (Button Bar)	29
Figure 2-2	NQE Status Window	29
Figure 2-3	NQE Configuration Information Window	30
Figure 2-4	NQE Submit Window	31
Figure 2-5	Typical Display from ArrayView (aview) Command	40
Figure 3-1	IRISconsole Main Window	48
Figure 3-2	IRISconsole Systems Window	49
Figure 3-3	IRISconsole Hardware Status Window	50
Figure 3-4	IRISconsole System Console Window	51
Figure 4-1	Gaining Efficiency Through Asynchronous Communication	75
Figure A-1	RendAsunder Graphics Windows	87
Figure A-2	RendAsunder Controls Menu	88

List of Tables

Table i	Typographical Conventions	xv
Table 1-1	Array Node System Selection	5
Table 1-2	Information Sources: Array Component Systems	6
Table 1-3	Information Sources: HIPPI Interconnect	7
Table 1-4	Information Sources: IRISconsole	9
Table 1-5	Information Sources: Management Tools	13
Table 1-6	Information Sources: Interactive Processing	14
Table 1-7	Information Sources: Batch Scheduling Products	16
Table 1-8	Information Sources: Compilers from SGI	17
Table 1-9	Information Sources: High Performance Fortran	18
Table 1-10	Information Sources: CHALLENGEcomplib	19
Table 1-11	Information Sources: IRIX and REACT	20
Table 1-12	Information Sources: Performance and Debugging Tools	20
Table 1-13	Information Sources: Parallel and Distributed Programming	22
Table 2-1	Information Sources: Invoking a Program	26
Table 2-2	Information Sources: Local Process Management	27
Table 2-3	NQE Command Line Interface Summary	32
Table 2-4	Information Sources: Array Services Commands	33
Table 2-5	Array Services Command Option Summary	35
Table 2-6	Array Services Environment Variables	36
Table 3-1	Information Sources: Console Management	52
Table 3-2	Information Sources: Array Configuration	53
Table 3-3	Subentries of a COMMAND Definition	60
Table 3-4	Substitutions Used in a COMMAND Definition	60
Table 3-5	Options of the COMMAND Definition	61
Table 3-6	Subentries of the LOCAL Entry	62
Table 4-1	Information Sources: General Numerical Libraries	67

Table 4-2	Information Sources: Libraries for Parallel Computation	67
Table 4-3	Information Sources: Performance Analysis Tools	68
Table 4-4	Information Sources: Software Pipelining	68
Table 4-5	Information Sources: Parallel Computation Models	70
Table 4-6	Array Services Data Structures	77
Table 4-7	Error Message Functions	78
Table 4-8	Functions for Connections to Array Services Daemons	78
Table 4-9	Server Options Functions Can Query or Change	79
Table 4-10	Functions for Interrogating the Configuration	80
Table 4-11	Functions for Managing Array Service Handles	80
Table 4-12	Functions for ASH Interrogation	81
Table A-1	Information Sources	90
Table B-1	Information Sources	91

About This Guide

This guide introduces the administration and programming features of in the Silicon Graphics, Inc.® Array 3.0 software product. Array 3.0 software supports Silicon Graphics, Inc. Array systems. Array systems are affordable, scalable systems that are used both by commercial users who require ultra-reliable database and file servers, and by scientific users who require high performance.

An Array system, as supported by Array 3.0, is a cluster of Silicon Graphics, Inc. systems connected by a network (for details, see “Array Nodes” on page 5). Array 3.0 software can also run on individual multiprocessors.

Specific hardware packages that have been sold as Array systems include POWER CHALLENGEarray™, CHALLENGE® DataArray, and other names. In this book, all systems on which Array 3.0 can run are collectively called Array systems.

Array systems are marketed to different audiences and used for different kinds of work, but the architectural concepts are the same, and so are the software tools used by system administrators and by software developers.

Audience

This guide is written for three groups of people: *users* who want to run software on an Array; *administrators* who need to configure an Array; and software *developers* who want to write programs for an Array.

An Array system is composed of many layers of hardware and software features, and each feature is documented separately. The main purpose of this guide is to help you orient yourself amid the profusion of printed and online documentation that is available.

If you are a system administrator, this guide introduces you to the information and software tools you use to configure an Array system, tune it, and keep it running.

If you are a software developer, this guide introduces you to the development tools and libraries you use to write programs that do high-performance parallel computation on an Array system.

Structure of This Guide

This guide contains the following chapters and appendices:

- Chapter 1, “Array System Components,” introduces the hardware and software architecture of the Array systems.
- Chapter 2, “Using an Array,” shows how users log in to an array, learn the array inventory and status, and manage interactive processes.
- Chapter 3, “Administering an Array,” covers the essential administrator tools for configuring Array Services and managing the Array system.
- Chapter 4, “Performance-Driven Programming in Array 3.0,” introduces the facilities for parallel programming including the MPI and PVM libraries.
- Appendix A, “The RendAsunder Demo Program,” describes an important demonstration program included with Array systems.
- Appendix B, “Array Documentation Quick Reference,” is a quick reference to all other Array information sources, including books, reference pages, and WWW sites. Use it as your online hypertext directory to information.

Conventions Used in This Guide

This manual uses the conventions and symbols shown in Table i.

Table i Typographical Conventions

Type of Information	How It Appears in the Text
Filenames and pathnames	This structure is declared in <i>/usr/include/sys/time.h</i> .
IRIX command names and options used in normal text	Update these variables with <i>sys tune</i> ; then build a new kernel with <i>autoconfig -vf</i> .
Names of program variables, structures, and data types	Global variable <i>mainSema</i> points to an IRIX semaphore, which has type <i>usema_t</i> .
Names of IRIX kernel functions, library functions, and functions in example code	Use mmap() to map an object into the address space, and munmap() to remove it.
User input within a multiline example	<code>tokyo% ainfo ash</code>

Array System Components

An Array system is a complex system with layers of hardware and software components. This chapter orients you to these components, working from the bottom up:

- “Array Components” on page 2 surveys the hardware parts—high performance “nodes,” network switches, and so on—and software that compose an Array.
- “Array Architecture” on page 5 surveys the way the Array is connected into a single system by a high-speed network and Array 3.0 software.
- “Distributed Management Tools” on page 10 lists the special tools that an administrator can use to manage an Array, and also covers some tools that a system programmer can use for performance analysis.
- “Job Execution Facilities” on page 13 surveys the job entry and job management facilities available to Array system users.
- “Compilation, Development, and Execution Facilities” on page 16 describes the programming tools an Array system can provide.
- “Message-Passing Protocols” on page 21 is a survey of the facilities for distributed multiprocessing.

Each section contains a table of information sources—online and printed books, reference pages, and WWW sites—related to the topic of that section. All such pointers are reproduced in Appendix B, “Array Documentation Quick Reference.”

Array Components

The performance and power of an Array system are the result of linking several symmetric multiprocessor (SMP) computers by a high-performance interconnect, and managing the combination with customized software and bundled application and administrative software.

Array Hardware Components

An Array comprises the following hardware:

- From two to eight nodes, each of which is a Silicon Graphics, Inc. computer, typically a multiprocessor such as:
 - Origin2000™ or Origin200™
 - Challenge®, POWER Challenge™, or POWER Challenge R10000™
 - Onyx2™, Onyx™ or POWER Onyx™
- An interconnecting network, typically one, and as many as six, bidirectional HIPPI network interfaces per node and a HIPPI crossbar switch.
- One IRISconsole™ as an administration console.
An IRISconsole is an O2™ or Indy™ workstation augmented with an IRISconsole serial port multiplexer.

A complete Array system is shown schematically in Figure 1-1.

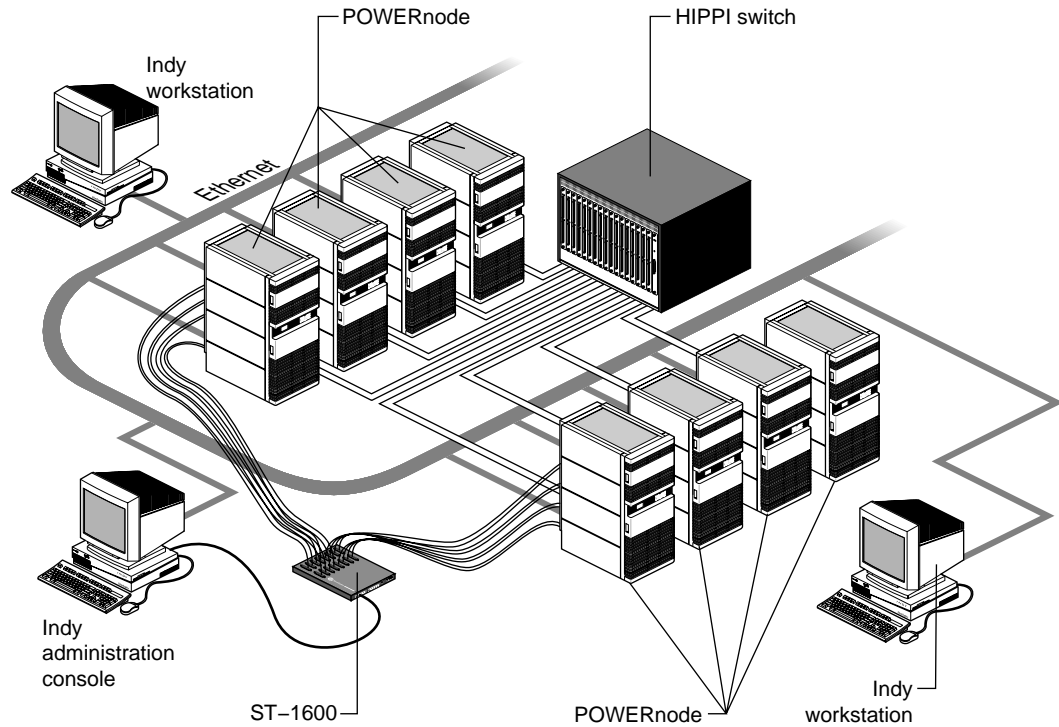


Figure 1-1 Array System Schematic

Array Software Components

The Array 3.0 software binds the Array system hardware into a supercomputer that can be programmed and administered as one system. An Array system using Array 3.0 software is based on the following major components:

Array diagnostics	Diagnostics used by Silicon Graphics, Inc. system engineers to verify installation and isolate faults.
IRIX™ 6.2 and IRIX 6.4	Multiprocessor operating systems including NFS™ version 3 network support.
XFS™ filesystem	High-performance, ultra-high capacity, journaled filesystem manages large RAID and disk farms.

HIPPI software	Support for high-performance network link, including an SGI-proprietary fast path for minimum overhead on short messages.
Array Services	Integrated administration tools.
IRISconsole	Permits centralized administration of all nodes in the Array.
MPI (Message-Passing Interface) 3.0 and XMPI	Distributed programming environment with support for HIPPI bypass, and visual monitor.
PVM (Parallel Virtual Machine) 1.2 and XPVM	Popular distributed programming environment and visual monitor.

Many optional software packages are available from Silicon Graphics, Inc. to extend Array 3.0, including:

Network Queuing Environment (NQE)	Load-balancing and scheduling facility that lets users submit, monitor, and control work across machines in a network.
Performance Co-Pilot™ (PCP)	Performance visualization facility.
ProDev™ WorkShop	Suite of graphical tools for developing parallel programs.

A variety of software packages from third parties also are available, including:

SHARE II™	Resource-centric Fair Share™ scheduler from Softway (systems using IRIX 6.2 only).
PerfAcct™	Accounting software by Instrumental, Inc.
Codine™	Batch-scheduling facility by GENIAS Software.
LSF™ (Load Sharing Facility)	Batch scheduling facility by Platform Computing.
High Performance Fortran (HPF)	Compilers available from the Portland Group™ (PGI) and Applied Parallel Research™ (APR)

Most of these components are described at more length in following topics.

Array Architecture

An Array system is a distributed-memory multiprocessor, scalable to several hundred individual MIPS processors in as many as eight nodes, yielding a peak aggregate computing capacity of many GFLOPS. The aggregation of nodes is connected by an industry-standard, 1.0 Gbit per second HIPPI network.

This section examines the components of an Array system in detail.

Array Nodes

The basic computational building block of an Array is a Silicon Graphics, Inc. multiprocessor. Any system running IRIX 6.2 can participate as a node in Array 3.0, but normally a node is a multiprocessor system. Depending on the type of Array and customer's choices, a node can be any of the systems listed in Table 1-1.

Table 1-1 Array Node System Selection

System	Processor Complement	Graphics
Origin2000	2-128 R10000	
Origin200	2 R10000	
Onyx2	2-16 R10000	InfiniteReality™ or RealityMonster™
CHALLENGE	2-32 R4400	
CHALLENGE 10000	2-36 R10000	Extreme™ Visualization Console
POWER Challenge	2-18 R8000	Extreme™ Visualization Console
POWER Challenge GR	2-24 R10000	Extreme™ Visualization Console, or InfiniteReality™, or Reality Engine2™
POWER Onyx	1-12 R8000	1-3 RealityEngine2™
Onyx 10000	1-24 R10000	1-3 InfiniteReality™

Table 1-2 lists information sources for the different types of systems.

Table 1-2 Information Sources: Array Component Systems

Topic	Book or URL	Book Number
All SGI Servers	http://www.sgi.com/Products/index.html?hardware	
Origin2000 and Origin200	http://www.sgi.com/Products/hardware/servers/index.html	
Onyx2 and RealityMonster	http://www.sgi.com/Products/hardware/graphics/products/index.html	
POWER CHALLENGE	<i>POWER CHALLENGE XL Rackmount Owner's Guide</i>	007-1735-xxx
POWER Onyx	<i>POWER Onyx and Onyx Rackmount Owner's Guide</i>	007-1736-xxx
CHALLENGE and CHALLENGE 10000	<i>POWER CHALLENGE XL Rackmount Owner's Guide</i>	007-1735-xxx
RealityEngine ² and InfiniteReality	http://www.sgi.com/Products/hardware/Onyx/Tech/	
Extreme Visualization Console	<i>POWER CHALLENGE XL Rackmount Owner's Guide</i>	007-1735-xxx

Hybrid Array

An Array that includes both Origin2000/Onyx2 systems and Challenge/Onyx systems is called a *hybrid* array. Previous versions of Array software supported only uniform Arrays composed of Challenge and Onyx systems. Array 3.0 software supports uniform arrays of Origin2000/Onyx2 systems, uniform arrays of Challenge/Onyx systems, and hybrid arrays.

The HIPPI Interconnect

Array nodes are normally connected by a high-performance, dual-channel HIPPI network. Each node is equipped with one or more bidirectional HIPPI interfaces. Each interface provides 100 MB per second of data bandwidth in either direction.

The HIPPI interfaces are connected via a high-performance HIPPI crossbar switch (optional in a two-node Array). The HIPPI switch is nonblocking, with sub-microsecond connection delays. The network appears to be fully connected and contention occurs only when two sources send data to the same destination at the same time.

IRIX 6.2 or IRIX 6.4 and the Array 3.0 software provide protocol layers and APIs to access the HIPPI network, including direct physical layer, HIPPI framing protocol, and TCP/IP. The HIPPI support includes special bypass capabilities to expedite transmission of short messages. The bypass capability is transparent to the applications using it.

Table 1-3 lists information sources on HIPPI and the HIPPI crossbar (which is produced by a third party, Essential Communications, Inc.).

Table 1-3 Information Sources: HIPPI Interconnect

Topic	Book or URL	Book Number
HIPPI interface	<i>IRIS HIPPI Administrator's Guide</i>	007-2229-xxx
	<i>IRIS HIPPI API Programmer's Guide</i>	007-2227-xxx
HIPPI Crossbar Switch	<i>EPS-1 User's Guide</i>	09-9010
	http://www.esscom.com	

Visualization and Interactive Supercomputing

Array nodes can be configured with hardware graphics support, to provide two and three-dimensional visualization performance commensurate with the available compute power. The available graphics options are listed in Table 1-1. Complex supercomputing visualization architectures can be built by aggregating compute and graphics nodes, as illustrated in Figure 1-2.

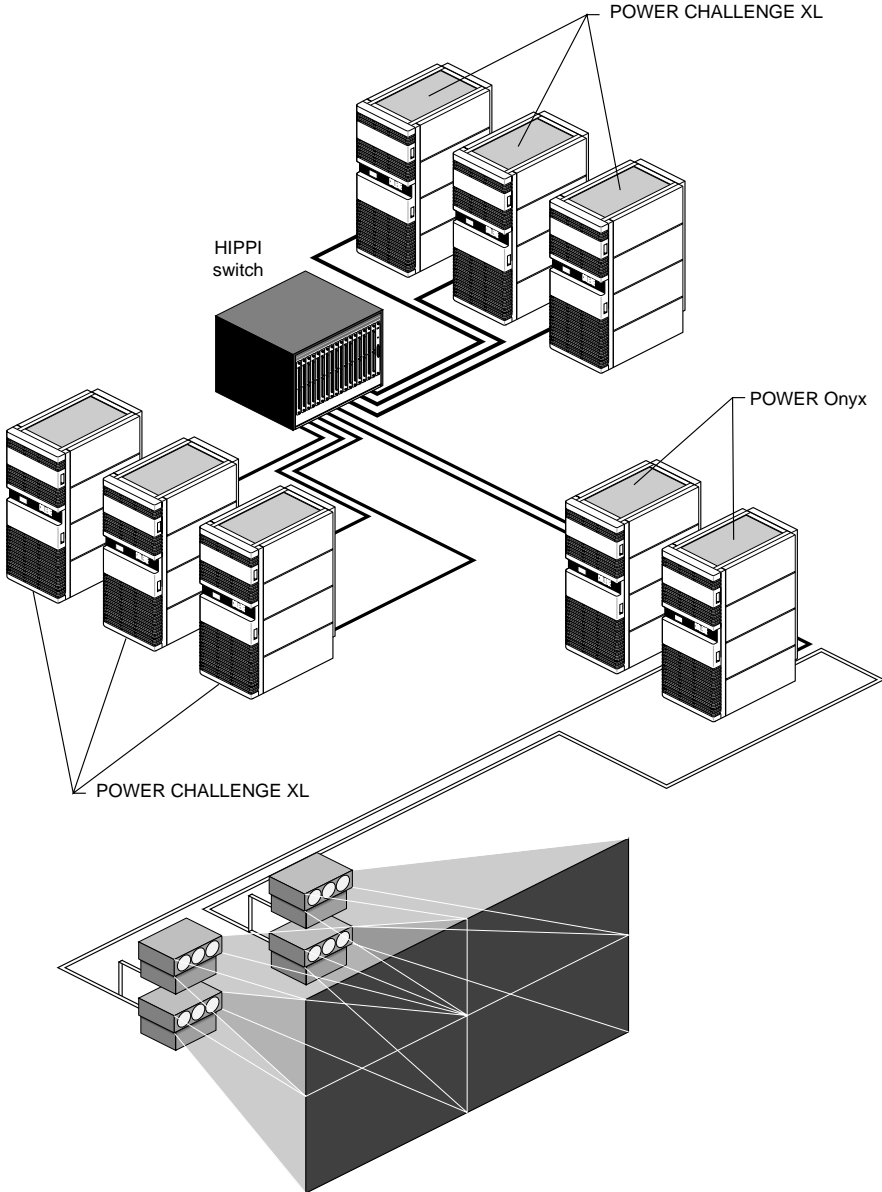


Figure 1-2 Advanced Visualization With Arrays

Centralized Console Management

An IRISconsole serves as a single, centralized administrative console for Array administration and maintenance. The IRISconsole consists of an O2 or Indy workstation, an IRISconsole multiplexer box, and the IRISconsole graphical cluster management software. From the IRISconsole, administrators can control, configure, monitor, and maintain the individual Array nodes.

The O2 or Indy workstation serves as a virtual console for each node. The workstation is connected to the multiplexer via a SCSI interface. The multiplexer in turn connects to the Remote System Control port of each node. Commands from the console workstation are routed to the appropriate node, and results from the nodes are routed back.

The IRISconsole graphical user interface provides a convenient graphic representation of the array. Sets of nodes can be selected and operated upon. You can open a command window directly to any node. You can use the IRISconsole graphical interface to

- Dynamically add and remove nodes in the array
- Display console messages or enter console commands to any node
- Interrupt, reset, or power-cycle any node
- Display and record real-time graphs of hardware operating statistics, including voltage, temperature, and cooling status
- Enable monitors and alarms for conditions such as excessive temperature
- View activity logs and other system reports

For more about the features of IRISconsole and illustrations of its use, see “Using the IRISconsole Workstation” on page 48. Table 1-4 lists other information sources for IRISconsole and its hardware.

Table 1-4 Information Sources: IRISconsole

Topic	Book or URL	Book Number
IRISconsole	<i>IRISconsole Administrator's Guide</i>	007-2872-xxx
	http://www.sgi.com/Products/hardware/challenge/IRISconsole.html	
IRISconsole hardware	<i>IRISconsole Installation Guide</i> <i>Indy Workstation Owner's Guide</i>	007-2839-xxx 007-9804-xxx

Distributed Management Tools

Array 3.0 makes an Array manageable by providing support for process execution, program development, performance instrumentation, and system administration.

This section introduces many of the bundled and third-party tools in detail.

Array Services

Array Services includes administrator commands, libraries, daemons and kernel extensions that support the execution of programs across an Array.

A central concept in Array Services is the *array session handle* (ASH), a number that is used to logically group related processes that may be distributed across multiple systems. The ASH creates a global process namespace across the Array, facilitating accounting and administration.

Array Services also provides an *array configuration database*, listing the nodes comprising an array. *Array inventory* inquiry functions provide a centralized, canonical view of the configuration of each node. Other *array utilities* let the administrator query and manipulate distributed array applications.

The Array Services package comprises the following primary components:

<i>array daemon</i>	These daemon processes, one in each node, cooperate to allocate ASH values and maintain information about node configuration and the relation of process IDs to ASHes.
<i>array configuration database</i>	One copy at each node, this file describes the Array configuration for use by array daemons and user programs.
<i>ainfo</i> command	Lets the user or administrator query the Array configuration database and information about ASH values and processes.
<i>array</i> command	Executes a specified command on one or more nodes. Commands are predefined by the administrator in the configuration database.
<i>arshell</i> command	Starts an IRIX command remotely on a different node using the current ASH value.
<i>aview</i> command	Displays a multiwindow, graphical display of each node's status.
<i>libarray</i> library	Library of functions that allow user programs to call on the services of array daemons and the array configuration database.

The use of the *ainfo*, *array*, *arshell*, and *aview* commands is covered in Chapter 2, “Using an Array.” The use of the *libarray* library is covered in Chapter 4, “Performance-Driven Programming in Array 3.0.”

Performance Co-Pilot

Performance Co-Pilot (PCP) is a Silicon Graphics product for monitoring, visualizing, and managing systems performance.

PCP has a distributed client-server architecture, with performance data collected from a set of servers and displayed on visualization clients. Performance data can be obtained from multiple sources, including the IRIX kernel and user applications. With support for low-intrusion performance data collection, reduction, and analysis, PCP permits a variety of metrics to be captured, correlated, reduced, recorded, and rendered.

PCP has been customized for Array systems to provide visualization of system-level and job-level statistics across the array. An array user can view a variety of relevant performance metrics on the array via the following utilities:

<i>mpvis</i>	Visualize CPU utilization of any node.
<i>dkvis</i>	Visualize disk I/O rates on any node.
<i>nfsvis</i>	Visualize NSF statistics on any node.
<i>pmchart</i>	Plot performance metrics versus time for any node.
<i>procvis</i>	Visualize CPU utilization across an array for tasks belonging to a particular array session handle.
<i>arrayvis</i>	Visualize aggregate Array performance.
<i>ashtop</i>	List of top CPU-using processes under a given ASH.
<i>arraytop</i>	List of top CPU-using processes in the array.

For more information about Performance Copilot, see *The Performance Co-Pilot User's and Administrator's Guide* (007-2614-xxx).

SHARE II (Fair Share) Scheduling

SHARE II, a “Fair Share” scheduler, allows an organization to create its own resource allocation policy based on its assessment of how resource usage should be fairly distributed to individuals or arbitrarily grouped users. SHARE II is available only for Arrays that use IRIX 6.2; it is not available for IRIX 6.4.

With SHARE II, users are grouped into a system-wide resource allocation and charging hierarchy. The hierarchy can represent projects, divisions, or arbitrary sets of users. Within this hierarchy, resource usage policy can be varied or delegated at any level according to organizational priorities.

Users can be limited in consumption of renewable resources (such as printer pages) and fixed resources (such as instantaneous memory use). Other limits are imposed during periods of scarcity (for example, CPU run time during periods of contention). Thus, SHARE II provides a fair share of the system resources during high-load periods without overcommitment, wasteful static reservations, or expensive administrator intervention.

Accounting With PerfAcct

PerfAcct, a third-party software product, gathers system accounting data from all nodes in a central location, where it is summarized and used to generate usage reports or billing. PerfAcct exploits IRIX extended-session accounting data to provide true job accounting. Job and project accounting permits usage tracking and billing by external or internal contracts, departments, tasks, and projects.

PerfAcct features low-overhead data collection on the nodes being monitored. To minimize system load on the monitored systems, archiving and summarization can be put on a remote low-cost workstation. PerfAcct also includes aggregate accounting statistics, as well as graphical user interface tools for measuring dynamic system load.

Supporting Documentation

Table 1-5 lists information sources for the management tool products.

Table 1-5 Information Sources: Management Tools

Topic	Book, Cross-Reference, or URL	Book Number
Array Services	Chapter 2, "Using an Array" array_services(5)	
Performance Co-Pilot data sheet	http://www.sgi.com/Products/hardware/challenge/CoPilot/CoPilot.html	
Performance Co-Pilot	<i>The Performance Co-Pilot User's and Administrator's Guide</i>	007-2614-xxx
	<i>Performance Co-Pilot for Informix-7 User's Guide</i>	007-3007-xxx
PerfAcct	http://www.instrumental.com	
SHARE II	<i>Share II for IRIX Administrator's Guide</i>	007-2622-xxx

Job Execution Facilities

An Array system can be used as an interactive system for real-time experimentation, as a coupled multiprocessor for grand-challenge class applications, and as a throughput compute engine for high-efficiency batch execution. This section introduces the job scheduling features.

Interactive Processing

Users can log in to a node to execute jobs interactively using normal IRIX job-control facilities. Interactive jobs can be command-line based, or can be X Windows applications that execute on the node but display on the user's workstation.

Jobs started interactively can be sequential programs, or multi-threaded programs executing within a node, or distributed-memory parallel applications executing across several nodes. Distributed programs using MPI or PVM can be started and monitored using the graphical monitors XMPI and XPVM; these display job status graphically on the user's workstation screen. Table 1-6 lists information sources on interactive processing.

Table 1-6 Information Sources: Interactive Processing

Topic	Book, Cross-Reference, or URL	Book Number
Logging in to a node	Chapter 2, "Using an Array"	
XMPI and XPVM	<i>MPI and PVM User's Guide</i> mpirun(1)	007-3286-xxx

Batch Processing

Batch processing allows off-line job scheduling. Batch processing is appropriate for production environments, high job-load environments, and situations where program results are not required immediately.

When an Array system is used for batch scheduling, users submit jobs to *batch queues*, which contain ordered sets of waiting jobs. When sufficient compute resources become available, and subject to tunable scheduling constraints, jobs are extracted from the batch queues and scheduled on the nodes. Job results and termination status are recorded in files or are electronically mailed to the user. See Figure 1-3.

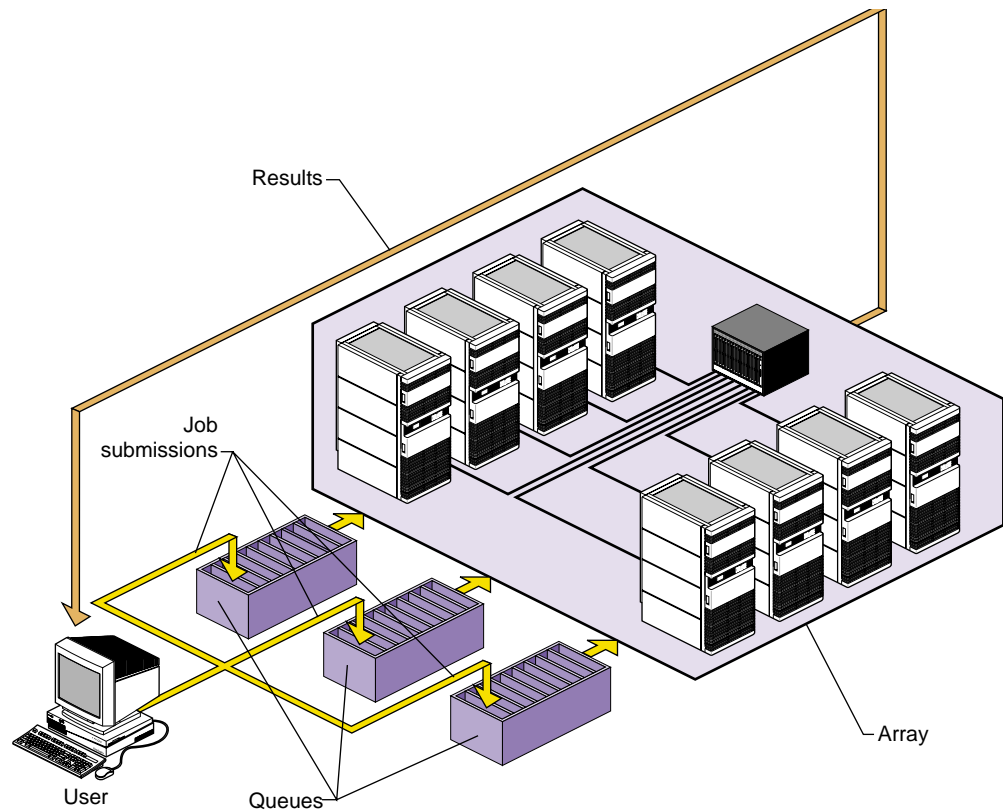


Figure 1-3 Batch Processing on an Array System

Several popular batch facilities are compatible with Array 3.0, including the Network Queuing Environment (NQE) from Silicon Graphics, Inc.; the Codine™ Job-Management System from Genias Software, Inc.; and Load Sharing Facility (LSF™) from Platform Computing, Inc.

NQE consists of the following components that provide a seamless environment for users of the Array:

- The NQE graphical interface allows users to submit batch requests to a central database, and to monitor and control each request.
- The Network Load Balancer (NLB) routes jobs to available nodes according to their current workload.

- The NQE scheduler determines when and on which node each request is to run.
- The File Transfer Agent (FTA) provides synchronous and asynchronous transfer of files, including automatic retry when a network link fails.

IRIX Checkpoint and Restart (CPR) facility allows you to save the status of long-running jobs and restart them easily.

Table 1-7 lists information sources on these products.

Table 1-7 Information Sources: Batch Scheduling Products

Topic	Book or URL	Book Number
IRIX Checkpoint and Restart (CPR)	<i>IRIX Checkpoint and Restart Operation Guide</i>	007-3236-xxx
Network Queuing Environment (NQE) technical papers	http://www.sdiv.cray.com/~nqe/nqe_external/index.html (pointers to technical papers)	
	http://www.cray.com/PUBLIC/product-info/sw/nqe/nqe30.html (illustrated overview)	
	<i>NQE User's Guide</i>	SG-2148 3.2
	<i>NQE Administrator's Guide</i>	SG-2150 3.2
Load Sharing Facility (LSF)	http://www.platform.com	
Codine	http://www.instrumental.com	

Compilation, Development, and Execution Facilities

Array 3.0 is complemented by development tools from Silicon Graphics, Inc. and other companies to simplify creation of parallel applications using both shared-memory and distributed-memory models. This section summarizes these tools. Additional discussion of software development appears in Chapter 4, "Performance-Driven Programming in Array 3.0."

Optimizing and Parallelizing Compilers

The MIPSpro™ compilers are the third-generation family of optimizing and parallelizing compilers from Silicon Graphics, offering comprehensive support for parallel application development.

Exploiting aggressive dependency analysis, the compilers perform automatic program restructuring, software pipelining, and parallelization. The compilers also provide a comprehensive set of comment directives that enable users to assist the compiler in the parallelization process.

Silicon Graphics, Inc. offers MIPSpro compilers for Fortran 77, Fortran 90, and C; as well as compilers for Ada 95, C++, assembly language, and Pascal. For detailed information about each compiler see the sources listed in Table 1-8.

Table 1-8 Information Sources: Compilers from SGI

Topic	Book or URL	Book Number
MIPSpro compiler features and use	<i>MIPS Compiling and Performance Tuning Guide</i>	007-2479-xxx
C language	<i>C Language Reference Manual</i>	007-0701-xxx
MIPSpro Fortran 77	<i>MIPSpro Fortran 77 Programmer's Guide</i>	007-2361-xxx
	<i>MIPSpro Fortran 77 Language Reference Manual</i>	007-2362-xxx
MIPSpro Fortran 90	<i>MIPSpro Fortran 90 Programmer's Guide</i>	007-2761-xxx
Automatic parallelization of C and Fortran code	<i>MIPSpro Power Fortran 77 Programmer's Guide</i>	007-2363-xxx
	<i>MIPSpro Power Fortran 90 Programmer's Guide</i>	007-2760-xxx
	<i>IRIS Power C User's Guide</i>	007-0702-xxx
C++ language	<i>C++ Programmers Guide</i>	007-0704-xxx
Assembly Language	<i>MIPSPro Assembly Language Programmer's Guide</i>	007-2418-xxx
Ada95 (GNU Ada Translator, GNAT)	<i>GNAT User's Guide</i>	007-2624-xxx
Pascal	<i>Pascal Programming Guide</i>	007-0740-xxx

High Performance Fortran

High Performance Fortran (HPF) is an extended version of Fortran 90 that is emerging as a standard for programming of shared- and distributed-memory systems in the data-parallel style. HPF incorporates a data-mapping model and associated directives that allow a programmer to specify how data is logically distributed in an application. An HPF compiler interprets these directives to generate code that minimizes interprocessor communication in distributed systems and maximizes data reuse in all types of systems.

HPF compilers are available for Array systems from the Portland Group, Inc. and Applied Parallel Research, Inc. Table 1-9 lists information sources for these products.

Table 1-9 Information Sources: High Performance Fortran

Topic	Book or URL	Book Number
High Performance Fortran textbook	<i>The High Performance Fortran Handbook</i> , Koelbel, Loveman, Schreiber, Steele Jr., and Zosel; MIT Press, 1994 (http://www-mitpress.mit.edu/)	ISBN 0-262-61094-9
High Performance Fortran forum	http://www.crpc.rice.edu/HPFF/home.html	
Portland Group, Inc.	http://www.pgroup.com	
Applied Parallel Research	http://www.infomall.org/apri	

Numerical Libraries

The compilers are complemented by CHALLENGEcomplib™, a comprehensive, optimized collection of scientific and math subroutine libraries popular in scientific computing. The library consists of two subcomponents: SGIMATH and SLATEC.

SGIMATH is hand-tuned, optimized, and parallelized, providing high-performance, portable implementations of the following popular numerical facilities:

- Basic Linear Algebra Subprograms (BLAS), levels 1, 2, and 3
- 1D, 2D, and 3D Fast Fourier Transforms (FFT)
- convolutions and correlation routines

- LAPACK, LINPACK, and EISPACK
- SCIPOPT (portable version of SCILIB)
- SOLVERS: pcg sparse solvers, direct sparse solvers, symmetric iterative solvers, and solvers for special linear systems

A source for a more detailed overview of CHALLENGEcomplib is listed in Table 1-10. Most of the functions within the library are documented in reference pages that install with the product.

Table 1-10 Information Sources: CHALLENGEcomplib

Topic	Book or URL	Book Number
CHALLENGEcomplib overview	http://www.sgi.com/Products/hardware/Power/ch_complib.html	

IRIX 6.2 and 6.4

The primary process control services of the Array are provided by the IRIX operating system, a symmetric multiprocessing operating system based on UNIX SVR4 with compatibility for BSD.

IRIX version 6.2 is required for Array 3.0 on Challenge/Onyx systems, and IRIX 6.4 on Origin systems. This version provides fast, flexible support for shared-memory interprocess communication, high-performance I/O, and performance-centric scheduling. Within a node, related processes are gang-scheduled to prevent one process from wasting time by spinning on locks held by blocked peers. Process placement decisions incorporate *cache affinity* heuristics, which minimizes multiprogramming-induced cache thrashing by tending to keep particular processes on the same processor.

Real-time processing can be supported with the REACT facilities, including nondegrading priorities, deadline scheduling, and reliably bounded kernel latencies. Hooks to support optional SHARE II Fair Share Scheduling checkpoint-restart facilities are also supported.

IRIX supports a variety of system functions to allow shared memory interprocess communication (IPC) between processes within one node. SVR4-compatible library functions for semaphores, message queues, and shared memory are supported.

High-performance IRIX-unique facilities for shared memory, semaphores, and mutex locks are included. POSIX-compatible library functions for semaphores, message queues, and shared memory are integrated into IRIX 6.4 (available as a patch set for IRIX 6.2).

Overview sources on IRIX and on the REACT real-time programming extensions are listed in Table 1-11.

Table 1-11 Information Sources: IRIX and REACT

Topic	Book or URL	Book Number
IRIX 6.2 Data Sheet	http://www.sgi.com/Products/software/IRIX6.2/IRIX62DS.html	
IRIX 6.2 Specifications	http://www.sgi.com/Products/software/IRIX6.2/IRIX62specs.html	
REACT/pro and real-time programming	http://www.sgi.com/real-time/	
IRIX IPC facilities	<i>Topics In IRIX Programming</i>	007-2478-xxx

Performance and Debugging Tools

Silicon Graphics includes a powerful set of parallel debugging, profiling, and visualization tools as part of the Developer Magic™ application development suite. Systemic performance visualization is provided by the Performance Co-Pilot facility, and array extensions.

In addition to these, IRIX 6.2 contains the interactive debugger dbx and profiling tools pixie and prof. Information sources on developer tools are listed in Table 1-12.

Table 1-12 Information Sources: Performance and Debugging Tools

Topic	Book or URL	Book Number
Developer Magic overview	http://www.sgi.com/Products/DevMagic/	
Developer Magic	<i>Developer Magic: ProDev WorkShop Overview</i> http://www.sgi.com/Products/WorkShop.html	007-2582-xxx

Table 1-12 (continued) Information Sources: Performance and Debugging Tools

Topic	Book or URL	Book Number
Performance Co-Pilot data sheet	http://www.sgi.com/Products/hardware/challenge/CoPilot/CoPilot.html	
Performance Co-Pilot	<i>The Performance Co-Pilot User's and Administrator's Guide</i>	007-2614-xxx
	<i>Performance Co-Pilot for Informix-7 User's Guide</i>	007-3007-xxx
<i>dbx, prof, pixie</i>	<i>dbx User's Guide</i>	007-0906-xxx
	<i>MIPS Compiling and Performance Tuning Guide</i>	007-2479-xxx

Message-Passing Protocols

Parallel applications using IPC facilities execute within a single node. However, you can create parallel applications that distribute across one or more nodes using a different model of parallel computation, the message-passing model.

In the message-passing model, processes communicate by exchanging “messages” of application data. The supporting library code chooses the fastest available means to pass the messages—through shared memory IPC within a node, across the HIPPI interconnect between nodes when available, or via TCP/IP.

Array 3.0 supports multiple message-passing protocols that are bundled in the separate product, the Message-Passing Toolkit. This single product contains implementations of three protocols: Two well-known, standardized, message-passing libraries, the *Message Passing Interface* (MPI) and *Parallel Virtual Machine* (PVM), and the Cray-designed SHMEM protocol.

MPI is the favored message-passing facility under Array 3.0. The MPI library exploits low-overhead, shared-memory transfers whenever possible. Messages sent between processes residing on different nodes use the HIPPI network; but the MPI library is aware of, and uses, the proprietary HIPPI bypass in Array 3.0 to get higher bandwidth when possible.

While Array 3.0 supports MPI as its native message-passing model, it also supports PVM and SHMEM for portability. The PVM library support has been optimized to exploit shared-memory transfers within a single node, but it does not take advantage of HIPPI bypass, and thus may not achieve the inter-node bandwidth of MPI.

Table 1-13 lists information sources about parallel and distributed programming. This subject is also explored in more detail in Chapter 4, "Performance-Driven Programming in Array 3.0."

Table 1-13 Information Sources: Parallel and Distributed Programming

Topic	Book or URL	Book Number
Parallel Programming Models Compared	<i>Topics In IRIX Programming</i>	007-2478-xxx
Message Passing Toolkit (MPT) in general	http://www.cray.com/PUBLIC/product-info/sw/	
MPI Overview	mpi(5)	
MPI References	<i>Using MPI</i> , Gropp, Lusk, and Skjellum, MIT Press 1995 (http://www.mitpress.mit.edu/) <i>MPI, The Complete Reference</i> , Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press 1995 <i>Using MPI</i> (in IRIX Insight library)	ISBN 0-262-69184-1 ISBN 0-262-57104-8 007-2855-001
MPI Standard	http://www.mcs.anl.gov/mpi	
PVM Overview	pvm(1PVM)	
PVM Reference	<i>PVM: Parallel Virtual Machine</i> , Geist, Beguelin, Dongarra, Weicheng Jiang, Manchek, and Sunderam, MIT Press 1994 http://www.netlib.org/pvm3/book/pvm-book.html	ISBN 0-262-57108-0
PVM Home Page	http://www.epm.ornl.gov/pvm/pvm_home.html	
Porting PVM to MPI	<i>Topics In IRIX Programming</i>	007-2478-xxx

Using an Array

An Array system is an aggregation of nodes, which are IRIX servers bound together with a high-speed network and Array 3.0 software. Array users are IRIX users who enjoy the advantage of greater performance and additional services. Array users access the system with familiar commands for job control, login and password management, and remote execution.

Array 3.0 augments conventional IRIX facilities with additional services for array users and for array administrators. The extensions include support for global session management, array configuration management, batch processing, message passing, system administration, and performance visualization.

This chapter introduces the extensions for Array use, with pointers to more detailed information. (Appendix B, “Array Documentation Quick Reference,” summarizes all the pointers for quick access.) The principal topics are as follows:

- “Using an Array System” on page 24 summarizes what a user needs to know, and the main facilities a user has available.
- “Managing Local Processes” on page 26 reviews the conventional tools for listing and controlling processes within one node.
- “Managing Batch Jobs with NQE” on page 28 summarizes the use of the Network Queueing Environment.
- “Using Array Services Commands” on page 33 describes the common concepts, options, and environment variables used by the Array Services commands.
- “Interrogating the Array” on page 37 summarizes how to use Array Services commands to learn about the Array and its workload, with examples.
- “Managing Distributed Processes” on page 41 summarizes how to use Array Services commands to list and control processes in multiple nodes.

Using an Array System

As an ordinary user of an Array system you are an IRIX (that is, UNIX) user, with the additional benefit of being able to run distributed sessions on multiple nodes of the Array. You access the Array from either

- A workstation such as an SGI O2
- An X-terminal
- An ASCII terminal

In each case, you log in to one node of the Array in the way you would log in to any remote UNIX host. From a workstation or an X-terminal you can of course open more than one terminal window and log into more than one node.

Finding Basic Usage Information

In order to use an Array, you need the following items of information:

- The name of the Array.
You use this *arrayname* in Array Services commands.
- The login name and password you will use on the Array
You use these when logging in to the Array to use it.
- The hostnames of the array nodes.
Typically these names follow a simple pattern, often *arrayname1*, *arrayname2*, etc.
- Any special resource-distribution or accounting rules that may apply to you or your group under a job scheduling system.

You can learn the hostnames of the array nodes if you know the array name, using the *ainfo* command:

```
ainfo -a arrayname machines
```


Logging In to an Array

Each node in an Array is a Silicon Graphics, Inc. multiprocessor system such as an Origin2000. Each node has an associated hostname and IP network address. Typically, you use an Array by logging in to one node directly, or by logging in remotely from another host (such as the Array console or a networked workstation). For example, from a workstation on the same network, this command would log you in to the node named *hydra6*:

```
rlogin hydra6
```

For details of the `rlogin` command, see the reference page `rlogin(1)`.

The system administrators of your Array may choose to disallow direct node logins in order to schedule array resources. If your site is configured to disallow direct node logins, your administrators will be able to tell you how you are expected to submit work to the array—perhaps through remote execution software or batch queueing facilities.

Invoking a Program

Once you have access to an Array you can invoke programs of several classes:

- Ordinary (sequential) applications
- Parallel shared-memory applications within a node
- Parallel message-passing applications within a node
- Parallel message-passing applications distributed over multiple nodes (and possibly other servers on the same network running Array 3.0)

If you are allowed to do so, you can invoke programs explicitly from a logged-in shell command line; or you may use remote execution or a batch queueing system.

Programs that are X-windows clients must be started from an X server, either an X-terminal or a workstation running X Windows.

Some application classes may require input in the form of command line options, environment variables, or support files upon execution. For example:

- X client applications need the `DISPLAY` environment variable set to specify the X server (workstation or X-terminal) where their windows will display.

The `DISPLAY` variable is normally set automatically when you use `rlogin` from an SGI workstation.

- A multithreaded program may require environment variables to be set describing the number of threads.
For example, C and Fortran programs that use parallel processing directives test the `MP_SET_NUMTHREADS` variable.
- MPI and PVM message-passing programs may require support files to describe how many tasks to invoke on which nodes.

Some information sources on program invocation are listed in Table 2-1.

Table 2-1 Information Sources: Invoking a Program

Topic	Book, Reference Page, or URL	Book Number
Remote login	rlogin(1)	
Setting environment variables	environ(5), env(1)	
Starting MPI and PVM jobs	<i>MPI and PVM User's Guide</i>	007-3286-xxx

Managing Local Processes

Each IRIX process has a *process identifier* (PID), a number that identifies that process within the node where it runs. It is important to realize that a PID is local to the node; so it is possible to have processes in different nodes using the same PID numbers.

Within a node, processes can be logically grouped in *process groups*. A process group is composed of a parent process together with all the processes that it creates. Each process group has a *process group identifier* (PGID). Like a PID, a PGID is defined locally to that node, and there is no guarantee of uniqueness across the Array.

Monitoring Processes and System Usage

You query the status of processes using the IRIX command `ps`. To generate a full list of all processes on a local system, use a command such as

```
ps -elfj
```

You can monitor the activity of processes using the command `top` (an ASCII display in a terminal window) or `gr_top` (displays in a graphical window).

For a global picture of the state of one node you can use *gr_osview*. It displays a variety of resource use values as histograms or bar-graphs in a graphical window. The command *gmemusage* displays memory use by all applications in the node where you start it.

Scheduling and Killing Local Processes

You can start a process at a reduced priority, so that it interferes less with other processes, using the *nice* command. If you use the *cs* shell, specify */usr/bin/nice* to avoid the built-in shell command *nice*. To start a whole shell at low priority, use a command like

```
/usr/bin/nice /bin/sh
```

You can schedule commands to run at specific times using the *at* command. You can kill or stop processes using the *kill* command. To destroy the process with PID 13032, use a command such as

```
kill -KILL 13032
```

Summary of Process Management Commands

Table 2-2 summarizes information about local process management.

Table 2-2 Information Sources: Local Process Management

Topic	Book, Reference Page, or URL	Book Number
Process ID and process group	intro(2) — scan to the section headed “Definitions”	
Listing and Monitoring Processes	ps(1), top(1), and gr_top(1); gr_osview(1), gmemusage(1)	
Running programs at low priority	nice(1), batch(1)	
Running programs at a scheduled time	at(1)	
Terminating a process	kill(1)	

Managing Batch Jobs with NQE

The Network Queuing Environment (NQE) is used to manage batch jobs. A batch job is a set of commands—a shell script. You submit batch job requests from a workstation to NQE, and NQE routes the jobs to an appropriate server. When a job completes, NQE returns the standard output and standard error files to the workstation. You can monitor the status of jobs, as well as delete or signal jobs.

NQE provides reliable file transfer with the File Transfer Agent (FTA), so that job scripts can transfer files to and from remote systems. If a file transfer fails for a transient reason such as a network link failing, FTA automatically requeues the transfer. This is useful in job requests because a job does not abort if the file transfer fails on the first attempt. If allowed by the site, a password is not required for the file transfer. This capability of FTA is called Network Peer-to-Peer Authorization (NPPA).

Accessing the NQE Commands

NQE is usually installed in `/usr/craysoft/nqe/bin` on IRIX workstations. If that directory doesn't exist, contact your system administrator to see if NQE is installed and where it is installed. If `/usr/craysoft/nqe/bin` does exist, add it to your PATH variable. For example:

```
% setenv PATH $PATH:/usr/craysoft/nqe/bin
```

or

```
$ export PATH=$PATH:/usr/craysoft/nqe/bin
```

Starting NQE

The easiest way to start using NQE is through its graphical interface as implemented by the `nqe` command (see the `nqe(1)` reference page). If you run `nqe` on your workstation, you just start it. If you need to start `nqe` on an array node, with output to your workstation, you may need to set your DISPLAY variable first, as shown in the following example:

```
% setenv DISPLAY myworkstation:0  
% nqe
```

Figure 2-1 shows the initial (top-level) NQE button bar window that should immediately appear.



Figure 2-1 NQE Top-level Window (Button Bar)

Checking Job Status with NQE

To see the status of jobs running under NQE, click on the Status button. Figure 2-2 shows an example of the Status window.

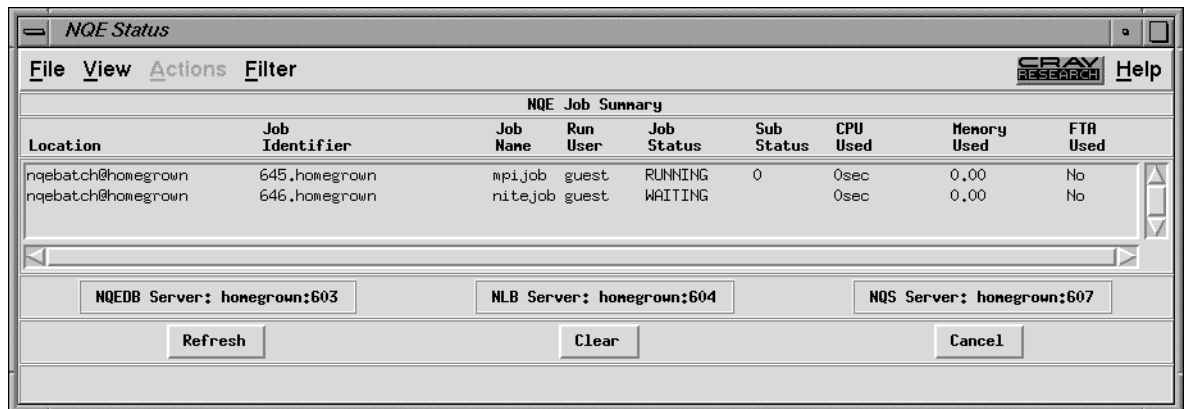


Figure 2-2 NQE Status Window

The example Status Window displays two jobs. Both are executing on the server *homegrown* and both are running (or will run) under the user account *guest*.

To refresh the status display, use the Refresh button in the Status window. You may also have the display refreshed periodically by setting the refresh option in the NQE Configuration Information Window, shown in Figure 2-3. Access the NQE Configuration Information Window using the Config button on the NQE button bar.

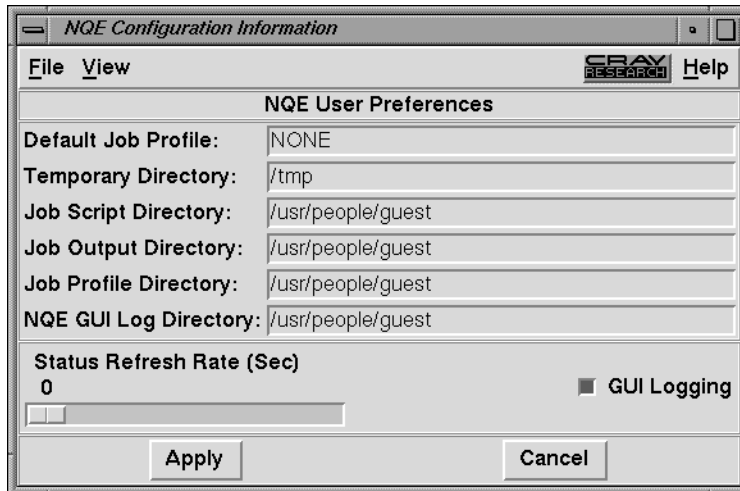


Figure 2-3 NQE Configuration Information Window

The slide bar titled “Status Refresh Rate” (in Figure 2-3) sets the refresh rate to a value other than 0. If the rate is set to 60, the NQE status display will be refreshed every 60 seconds.

Submitting a Job with NQE

To submit a new batch job, display the Submit window (accessed using the Submit button in the NQE button bar). Figure 2-4 shows an example of the Submit window with a sample job script. To submit the job, click the Submit button.

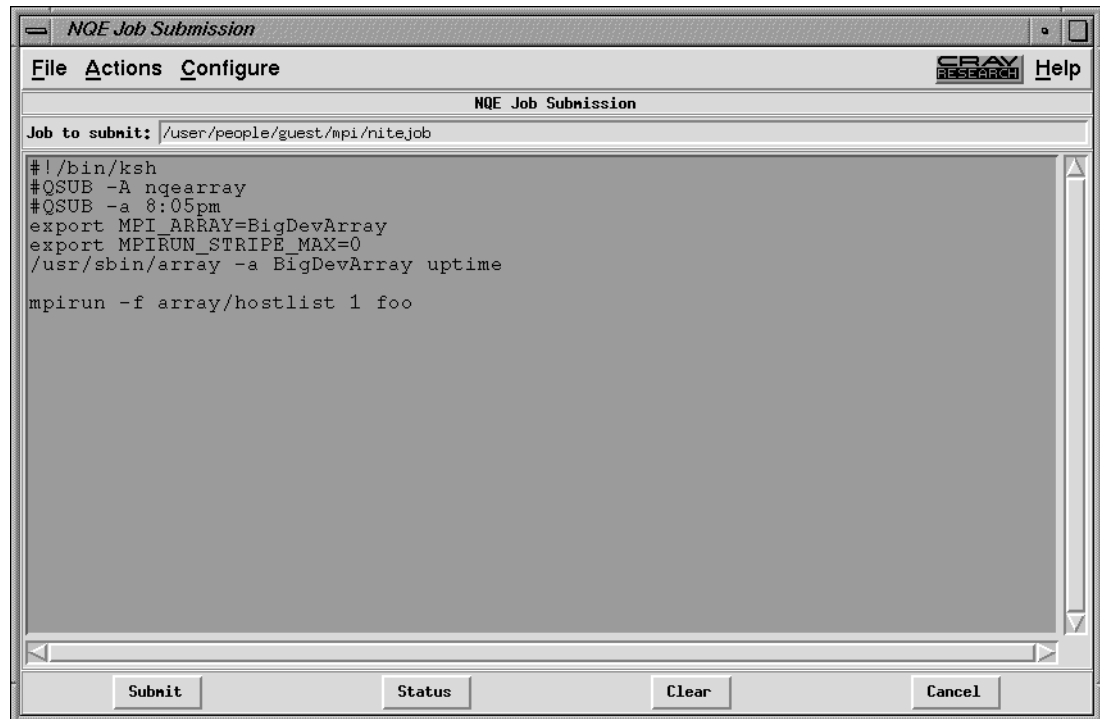


Figure 2-4 NQE Submit Window

A few details of the example job script shown in Figure 2-4 are of interest. The `#QSUB` string is an NQE directive, used to embed command line options within the script. (See the `cqsub(1)` or `qsub(1)` reference page for more information on embedded options.) The line

```
#QSUB -a 8:05
```

indicates to NQE that the job request should be started sometime after 8pm (20:00). The line

```
#QSUB -A nqearray
```

indicates to NQE that the job should run using the project “nqearray”. (See the `projects(5)` reference page for more information on project names.)

About NQE Command Line Interfaces

You can also operate NQE using a command-line interface. The NQE commands are summarized in Table 2-3. For details of the command-line interface, see the *NQE User's Guide*.

Table 2-3 NQE Command Line Interface Summary

Command Name	Purpose
cevent	Posts, reads, and deletes information on job-dependency events.
cqdel	Signals or deletes a job request.
cqstatl	Displays the status of job requests.
cqsub	Submits a job script.
ftua	File transfer utility, similar to FTP but with file transfer queuing and recovery (server command only).
ilb	Executes commands interactively on a host chosen by NQE.
qalter	Alters the attributes of a job request (server command only).
qchkpnt	Checkpoints a job (may only be invoked within a job script).
qdel	Signals or deletes a job request (server command only).
qlimit	Displays the job limits that apply to an NQE server (server command only).
qmsg	Writes messages to stderr, stdout, or the job log (server command only).
qping	Determines if the local NQS daemon is running (server command only).
qstat	Displays the status of job requests (server command only).
qsub	Submits a job script (server command only).
rft	File transfer command, suitable for use in job scripts (server command only).

Using Array Services Commands

When an application starts processes on more than one node, the PID and PGID are no longer adequate to manage the application. The commands of the Array Services component of Array 3.0 give you the ability to view the entire Array, and to control the processes of multinode programs.

Tip: You can use Array Services commands from any workstation connected to an Array system, for example from a workstation. You don't have to be logged in to an Array node.

This topic introduces the terms, concepts, and command options that are common to all Array Services commands. For details about any of the commands, see one of the reference pages listed in Table 2-4.

Table 2-4 Information Sources: Array Services Commands

Topic	Book, Reference Page, or URL	Book Number
Array Services Overview	array_services(5)	
<i>ainfo</i> command	ainfo(1)	
<i>array</i> command	use: array(1); configuration: arrayd.conf(4)	
<i>arshell</i> command	arshell(1)	
<i>aview</i> command	aview(1)	
<i>newsess</i> command	newsess(1)	

About Array Sessions

As noted under “Distributed Management Tools” on page 10, Array Services is composed of a daemon—a background process that is started at boot time in every node—and a set of commands such as *ainfo*. The commands call on the daemon process in each node to get the information they need.

One concept that is basic to Array Services is the *array session*, which is a term for all the processes of one application, wherever they may execute. Normally, your login shell, with the programs you start from it, constitutes an array session. A batch job is an array session; and you can create a new shell with a new array session identity.

Each session is identified by an *array session handle* (ASH), a number that identifies any process that is part of that session. You use the ASH to query and to control all the processes of a program, even when they are running in different nodes.

About Names of Arrays and Nodes

Each node is an IRIX server, and as such has a hostname. The hostname of a node is returned by the *hostname* command executed in that node:

```
% hostname
tokyo
```

The command is simple and documented in the *hostname(1)* reference page. The more complicated issues of *hostname* syntax, and of how hostnames are resolved to hardware addresses are covered in *hostname(5)*.

An Array system as a whole has a name too. In most installations there is only a single Array, and you never need to specify which Array you mean. However, it is possible to have multiple Arrays available on a network, and you can direct Array Services commands to a specific Array.

About Authentication Keys

It is possible for the Array administrator to establish an authentication code, which is a 64-bit number, for all or some of the nodes in an array (see “Configuring Authentication Codes” on page 58). When this is done, each use of an Array Services command must specify the appropriate authentication key, as a command option, for the nodes it uses. Your system administrator will tell you if this is necessary.

Summary of Common Command Options

The commands of Array Services—*ainfo*, *array*, *arshell*, *aview*, and *newsess*—have a consistent set of command options. Table 2-5 is a summary of these options. Not all options are valid with all commands; and each command has unique options besides those shown. The default values of some options are set by environment variables listed in the next topic.

Table 2-5 Array Services Command Option Summary

Option	Used In	Meaning
-a <i>array</i>	<i>ainfo, array, aview</i>	Specify a particular Array when more than one is accessible.
-D	<i>ainfo, array, arshell, aview</i>	Send commands to other nodes directly, rather than through array daemon.
-F	<i>ainfo, array, arshell, aview</i>	Forward commands to other nodes through the array daemon.
-Kl <i>number</i>	<i>ainfo, array, aview</i>	Authentication key (a 64-bit number) for the local node.
-Kr <i>number</i>	<i>ainfo, array, aview</i>	Authentication key (a 64-bit number) for the remote node.
-l (letter ell)	<i>ainfo, array</i>	Execute in context of the destination node, not the current node.
-p <i>port</i>	<i>ainfo, array, arshell, aview</i>	Nonstandard port number of array daemon.
-s <i>hostname</i>	<i>ainfo, array, aview</i>	Specify a destination node.

Specifying a Single Node

The *-l* and *-s* options work together. The *-l* (letter ell for local) option restricts the scope of a command to the node where the command is executed. By default, that is the node where the command is entered. When *-l* is not used, the scope of a query command is all nodes of the array. The *-s* (server, or node, name) option directs the command to be executed on a specified node of the array. These options work together in query commands as follows:

- To interrogate all nodes as seen by the local node, use neither option.
- To interrogate only the local node, use only *-l*.
- To interrogate all nodes as seen by a specified node, use only *-s*.
- To interrogate only a particular node, use both *-s* and *-l*.

Common Environment Variables

The Array Services commands depend on environment variables to define default values for the less-common command options. These variables are summarized in Table 2-6.

Table 2-6 Array Services Environment Variables

Variable Name	Use	Default When Undefined
ARRAYD_FORWARD	When defined with a string starting with the letter <i>y</i> , all commands default to forwarding through the array daemon (option <i>-F</i>).	Commands default to direct communication (option <i>-D</i>).
ARRAYD_PORT	The port (socket) number monitored by the array daemon on the destination node.	The standard number of 5434, or the number given with option <i>-p</i> .
ARRAYD_LOCALKEY	Authentication key for the local node (option <i>-Kl</i>).	No authentication unless <i>-Kl</i> option is used.
ARRAYD_REMOTEKEY	Authentication key for the destination node (option <i>-Kr</i>).	No authentication unless <i>-Kr</i> option is used.
ARRAYD	The destination node, when not specified by the <i>-s</i> option.	The local node, or the node given with <i>-s</i> .

Interrogating the Array

Any user of an Array system can use Array Services commands to check the hardware components and the software workload of the Array. The commands needed are *ainfo*, *array*, and *aview*.

Learning Array Names

If your network includes more than one Array system, you can use *ainfo arrays* at one array node to list all the Array names that are configured, as in the following example.

```
homegrown% ainfo arrays
Arrays known to array services daemon
ARRAY DevArray
    IDENT 0x3381
ARRAY BigDevArray
    IDENT 0x7456
ARRAY test
    IDENT 0x655e
```

Array names are configured into the array database by the administrator. Different Arrays might know different sets of other Array names.

Learning Node Names

You can use *ainfo machines* to learn the names and some features of all nodes in the current Array, as in the following example.

```
homegrown 175% ainfo -b machines
machine homegrown homegrown 5434 192.48.165.36 0
machine disarray disarray 5434 192.48.165.62 0
machine datarray datarray 5434 192.48.165.64 0
machine tokyo tokyo 5434 150.166.39.39 0
```

In this example, the *-b* option of *ainfo* is used to get a concise display.

Learning Node Features

You can use *ainfo nodeinfo* to request detailed information about one or all nodes in the array. To get information about the local node, use *ainfo -l nodeinfo*. However, to get information about only a particular other node, for example node *tokyo*, use *-l* and *-s*, as in the following example. (The example has been edited for brevity.)

```
homegrown 181% ainfo -s tokyo -l nodeinfo
Node information for server on machine "tokyo"
MACHINE tokyo
  VERSION 1.2
  8 PROCESSOR BOARDS
    BOARD: TYPE 15   SPEED 190
      CPU:   TYPE 9   REVISION 2.4
      FPU:   TYPE 9   REVISION 0.0
  ...
  16 IP INTERFACES  HOSTNAME tokyo   HOSTID 0xc01a5035
    DEVICE et0      NETWORK 150.166.39.0   ADDRESS 150.166.39.39  UP
    DEVICE atm0     NETWORK 255.255.255.255 ADDRESS 0.0.0.0         UP
    DEVICE atm1     NETWORK 255.255.255.255 ADDRESS 0.0.0.0         UP
  ...
  0 GRAPHICS INTERFACES
  MEMORY
    512 MB MAIN MEMORY
    INTERLEAVE 4
```

If the *-l* option is omitted, the destination node will return information about every node that it knows.

Learning User Names and Workload

The IRIX commands *who*, *top*, and *uptime* are commonly used to get information about users and workload on one server. The *array* command offers Array-wide equivalents to these commands.

Learning User Names

To get the names of all users logged in to the whole array, use *array who*. To learn the names of users logged in to a particular node, for example *tokyo*, use *-l* and *-s*, as in the following example. (The example has been edited for brevity and security.)

```

homegrown 180% array -s tokyo -l who
joecd    tokyo      frummage.eng.sgi -tcsh
joecd    tokyo      frummage.eng.sgi -tcsh
benf     tokyo      einstein.ued.sgi. /bin/tcsh
yohn     tokyo      rayleigh.eng.sg vi +153 fs/procfs/prd
...

```

Learning Workload

Two variants of the *array* command return workload information. The array-wide equivalent of *uptime* is *array uptime*, as follows:

```

homegrown 181% array uptime
homegrown: up 1 day, 7:40, 26 users, load average: 7.21, 6.35, 4.72
disarray:  up 2:53, 0 user, load average: 0.00, 0.00, 0.00
datarray:  up 5:34, 1 user, load average: 0.00, 0.00, 0.00
tokyo:    up 7 days, 9:11, 17 users, load average: 0.15, 0.31, 0.29
homegrown 182% array -l -s tokyo uptime
tokyo:    up 7 days, 9:11, 17 users, load average: 0.12, 0.30, 0.28

```

The command *array top* lists the processes that are currently using the most CPU time, with their ASH values, as in the following example.

```

homegrown 183% array top

```

ASH	Host	PID	User	%CPU	Command
0x1111ffff00000000	homegrown	5	root	1.20	vfs_sync
0x1111ffff000001e9	homegrown	1327	guest	1.19	atop
0x1111ffff000001e9	tokyo	19816	guest	0.73	atop
0x1111ffff000001e9	disarray	1106	guest	0.47	atop
0x1111ffff000001e9	datarray	1423	guest	0.42	atop
0x1111ffff00000000	homegrown	20	root	0.41	ShareII
0x1111ffff000000c0	homegrown	29683	kchang	0.37	ld
0x1111ffff0000001e	homegrown	1324	root	0.17	arrayd
0x1111ffff00000000	homegrown	229	root	0.14	routed
0x1111ffff00000000	homegrown	19	root	0.09	pdflush
0x1111ffff000001e9	disarray	1105	guest	0.02	atopm

The *-l* and *-s* options can be used to select data about a single node, as usual.

Browsing With ArrayView

The *ArrayView*, or *aview*, command is a graphical window on the status of an array. You can start it with the command *aview* and it displays a window similar to the one shown in Figure 2-5. The top window shows one line per node. There is a window for each node, headed by the node name and its hardware configuration. Each window contains a snapshot of the busiest processes in that node.



Figure 2-5 Typical Display from ArrayView (aview) Command

Managing Distributed Processes

Using commands from the Array Services component of Array 3.0, you create and manage processes that are distributed across multiple nodes of the Array system.

About Array Session Handles (ASH)

In an Array system you can start a program whose processes are in more than one node. In order to name such collections of processes, Array 3.0 software assigns each process to an *array session handle* (ASH).

An ASH is a number that is unique across the entire array (unlike a PID or PGID). An ASH is the same for every process that is part of a single *array session*—no matter which node the process runs in. You display and use ASH values with Array Services commands. Each time you log in to an Array node, your shell is given an ASH, which is used by all the processes you start from that shell.

The command *ainfo ash* returns the ASH of the current process on the local node, which is simply the ASH of the *ainfo* command itself.

```
homegrown 178% ainfo ash
Array session handle of process 10068: 0x1111ffff000002c1
homegrown 179% ainfo ash
Array session handle of process 10069: 0x1111ffff000002c1
```

In the preceding example, each instance of the *ainfo* command was a new process: first PID 10068, then PID 10069. However, the ASH is the same in both cases. This illustrates a very important rule: *every process inherits its parent's ASH*. In this case, each instance of *array* was forked by the command shell, and the ASH value shown is that of the shell, inherited by the child process.

You can create a new global ASH with the command *ainfo newash*, as follows:

```
homegrown 175% ainfo newash
Allocating new global ASH
0x11110000308b2f7c
```

This feature has little use at present. There is no existing command that can change its ASH, so you cannot assign the new ASH to another command. It is possible to write a program that takes an ASH from a command-line option and uses the Array Services function *setash()* to change to that ASH (however such a program must be privileged). No such program is distributed with Array 3.0 (but see “Managing Array Service Handles” on page 80).

Listing Processes and ASH Values

The command `array ps` returns a summary of all processes running on all nodes in an array. The display shows the ASH, the node, the PID, the associated username, the accumulated CPU time, and the command string.

To list all the processes on a particular node, use the `-l` and `-s` options. To list processes associated with a particular ASH, or a particular username, pipe the returned values through `grep`, as in the following example. (The display has been edited to save space.)

```
homegrown 182% array -l -s tokyo ps | fgrep wombat
0x261cffff0000054c      tokyo 19007   wombat      0:00 -csh
0x261cffff0000054a      tokyo 17940   wombat      0:00 csh -c (setenv...
0x261cffff0000054c      tokyo 18941   wombat      0:00 csh -c (setenv...
0x261cffff0000054a      tokyo 17957   wombat      0:44 xem -geometry 84x42
0x261cffff0000054a      tokyo 17938   wombat      0:00 rshd
0x261cffff0000054a      tokyo 18022   wombat      0:00 /bin/csh -i
0x261cffff0000054a      tokyo 17980   wombat      0:03 /usr/gnu/lib/ema...
0x261cffff0000054c      tokyo 18928   wombat      0:00 rshd
```

When you have Performance Co-Pilot installed (see “Performance Co-Pilot” on page 11) you have two additional commands for listing processes: `ashtop` displays a continuously updated list of the processes that are executing under a specified ASH (see the `ashtop(1)` reference page, if installed). The `arraytop` command produces a similar display for the entire array (see the `arraytop(1)` reference page, if installed). Both of these, and additional features of Performance Co-Pilot, are described in the `pcp_array(5)` reference page.

Controlling Processes

The `arshell` command lets you start an arbitrary program on a single other node. The `array` command gives you the ability to suspend, resume, or kill all processes associated with a specified ASH.

Using arshell

The `arshell` command is an Array Services extension of the familiar `rsh` command; it executes a single IRIX command on a specified Array node. The difference from `rsh` is that the remote shell executes under the same ASH as the invoking shell (this is not true of simple `rsh`). The following example demonstrates the difference.

```
homegrown 179% ainfo ash  
Array session handle of process 8506: 0x1111ffff00000425  
homegrown 180% rsh guest@tokyo ainfo ash  
Array session handle of process 13113: 0x261cffff0000145e  
homegrown 181% arshell guest@tokyo ainfo ash  
Array session handle of process 13119: 0x1111ffff00000425
```

You can use *arshell* to start a collection of unrelated programs in multiple nodes under a single ASH; then you can use the commands described under “Managing Session Processes” on page 44 to stop, resume, or kill them.

Both MPI and PVM use *arshell* to start up distributed processes.

Tip: The shell is a process under its own ASH. If you use the *array* command to stop or kill all processes started from a shell, you will stop or kill the shell also. In order to create a group of programs under a single ASH that can be killed safely, proceed as follows:

1. Create a nested shell with a new ASH using *newsess*. Note the ASH value.
2. Within the new shell, start one or more programs using *arshell*.
3. Exit the nested shell.

Now you are back to the original shell. You know the ASH of all programs started from the nested shell. You can safely kill all jobs that have that ASH because the current shell is not affected.

About the Distributed Example

The programs launched with *arshell* are not coordinated (they could of course be written to communicate with each other, for example using sockets), and you must start each program individually.

The *array* command is designed to permit the simultaneous launch of programs on all nodes with a single command. However, array can only launch programs that have been configured into it, in the Array Services configuration file. (The creation and management of this file is discussed under “About Array Configuration” on page 53.)

In order to demonstrate process management in a simple way from the command line, the following command was inserted into the configuration file `/usr/lib/array/arrayd.conf`:

```
#
# Local commands
#
command spin                # Do nothing on multiple machines
    invoke /usr/lib/array/spin
    user   %USER
    group  %GROUP
    options nowait
```

The invoked command, `/usr/lib/array/spin`, is a shell script that does nothing in a loop, as follows:

```
#!/bin/sh
# Go into a tight loop
#
interrupted() {
    echo "spin has been interrupted - goodbye"
    exit 0
}
trap interrupted 1 2
while [ ! -f /tmp/spin.stop ]; do
    sleep 5
done
echo "spin has been stopped - goodbye"
exit 1
```

With this preparation, the command `array spin` starts a process executing that script on every processor in the array. Alternatively, `array -l -s nodename spin` would start a process on one specific node.

Managing Session Processes

The following command sequence creates and then kills a `spin` process in every node. The first step creates a new session with its own ASH. This is so that later, `array kill` can be used without killing the interactive shell.

```
homegrown 175% ainfo ash
Array session handle of process 8912: 0x1111ffff0000032d
homegrown 176% newsess
homegrown 175% ainfo ash
Array session handle of process 8941: 0x11110000308b2fa6
```

In the new session with ASH 0x11110000308b2fa6, the command `array spin` starts the `/usr/lib/array/spin` script on every node. In this test array, there were only two nodes on this day, *homegrown* and *tokyo*.

```
homegrown 176% array spin
```

After exiting back to the original shell, the command `array ps` is used to search for all processes that have the ASH 0x11110000308b2fa6.

```
homegrown 177% exit
homegrown 178% homegrown 177%
homegrown 177% ainfo ash
Array session handle of process 9257: 0x1111ffff0000032d
homegrown 179% array ps | fgrep 0x11110000308b2fa6
0x11110000308b2fa6 homegrown 9033 guest 0:00 /bin/sh /usr/lib/array/spin
0x11110000308b2fa6 homegrown 9618 guest 0:00 sleep 5
0x11110000308b2fa6 tokyo 26021 guest 0:00 /bin/sh /usr/lib/array/spin
0x11110000308b2fa6 tokyo 26072 guest 0:00 sleep 5
0x1111ffff0000032d homegrown 9642 guest 0:00 fgrep 0x11110000308b2fa6
```

There are two processes related to the *spin* script on each node. The next command kills them all.

```
homegrown 180% array kill 0x11110000308b2fa6
homegrown 181% array ps | fgrep 0x11110000308b2fa6
0x1111ffff0000032d homegrown 10030 guest 0:00 fgrep 0x11110000308b2fa6
```

The command `array suspend 0x11110000308b2fa6` would suspend the processes instead (however, it is hard to demonstrate that a *sleep* command has been suspended).

Administering an Array

An Array system is an aggregation of nodes, which are IRIX servers bound together with a high-speed network and Array 3.0 software. Array administrators are IRIX system administrators who must use additional tools to configure and manage the Array as an Array.

This chapter surveys the tools that you as an administrator use, with pointers to more detailed information. (Appendix B, “Array Documentation Quick Reference,” summarizes all pointers for quick access.) The main topics covered include:

- “Using the IRISconsole Workstation” on page 48 summarizes the look and feel of IRISconsole, and indicates some of the other tools you can also run at the console workstation.
- “About Array Configuration” on page 53 describes the use and contents of the Array Services database file, and the methods you use to change it and test the changes.
- “Configuring Arrays and Machines” on page 57 details how you define the Array and its nodes in the configuration files.
- “Configuring Authentication Codes” on page 58 summarizes authentication lines in the configuration database.
- “Configuring Array Commands” on page 58 details the way you define new subcommands for users to execute via the *array* command.

Using the IRISconsole Workstation

The Silicon Graphics, Inc. O2 or Indy workstation running IRISconsole is your primary control point, from which you can manage the hardware and software of all nodes in the array, performing all operator actions on every node, even including power-cycling and rebooting. Figure 3-1 shows the IRISconsole main window.



Figure 3-1 IRISconsole Main Window

From the main window you can open the Systems window in which you find an icon for each node in the selected Array. This is shown in Figure 3-2.

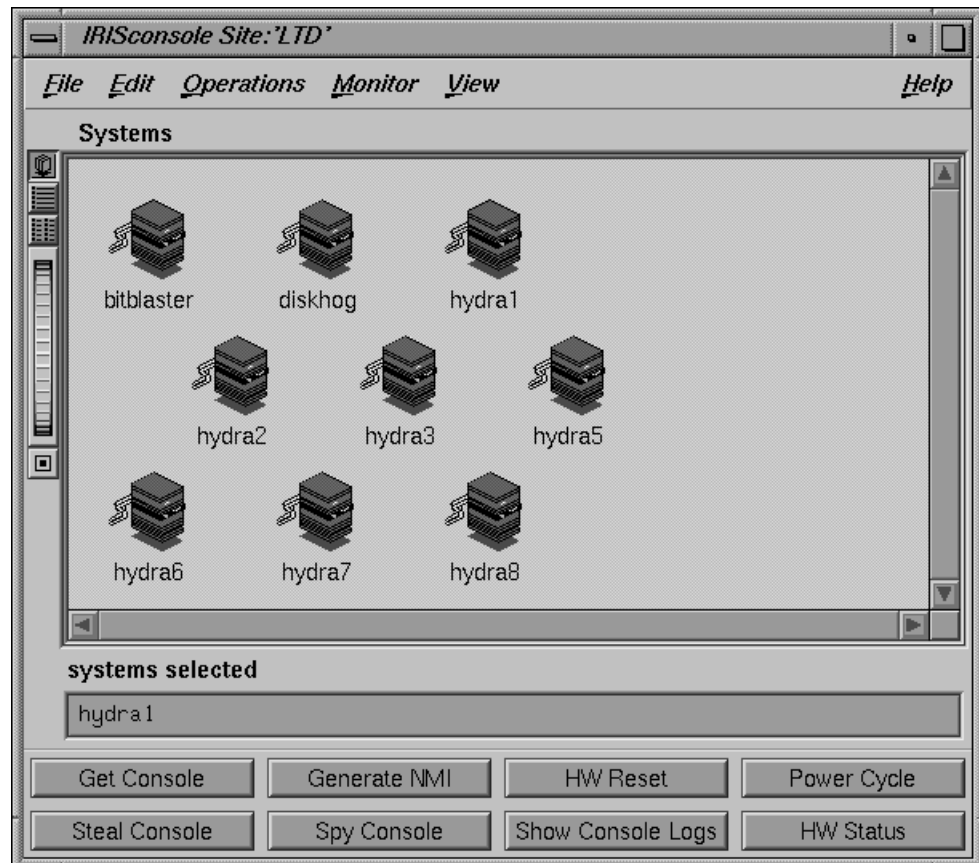


Figure 3-2 IRISconsole Systems Window

In the Systems window you can select one system icon and apply any of the functions shown as buttons to that system. Several functions allow you to perform radical hardware operations: power-cycle, reset, or cause a nonmaskable interrupt (NMI).

The HW Status function opens a hardware status window for the selected system, as shown in Figure 3-3. From this window you can monitor power-supply voltages, internal temperature, and fan RPM. (This feature is available only for nodes that are Challenge or Onyx systems, not for Origin2000 systems.)

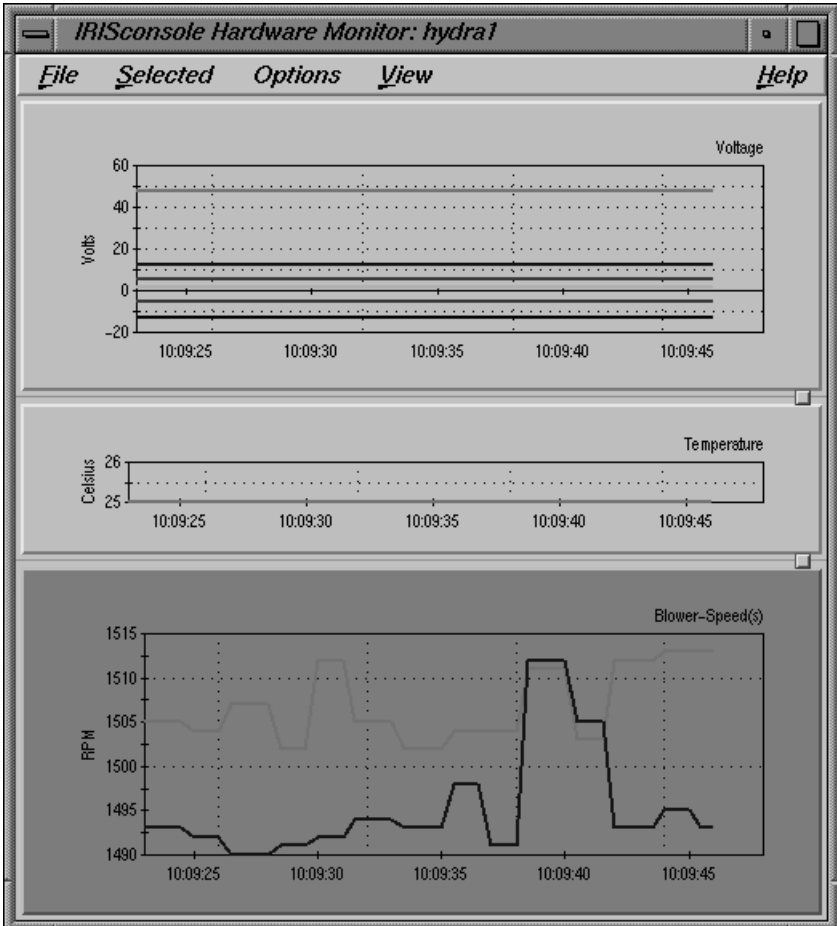
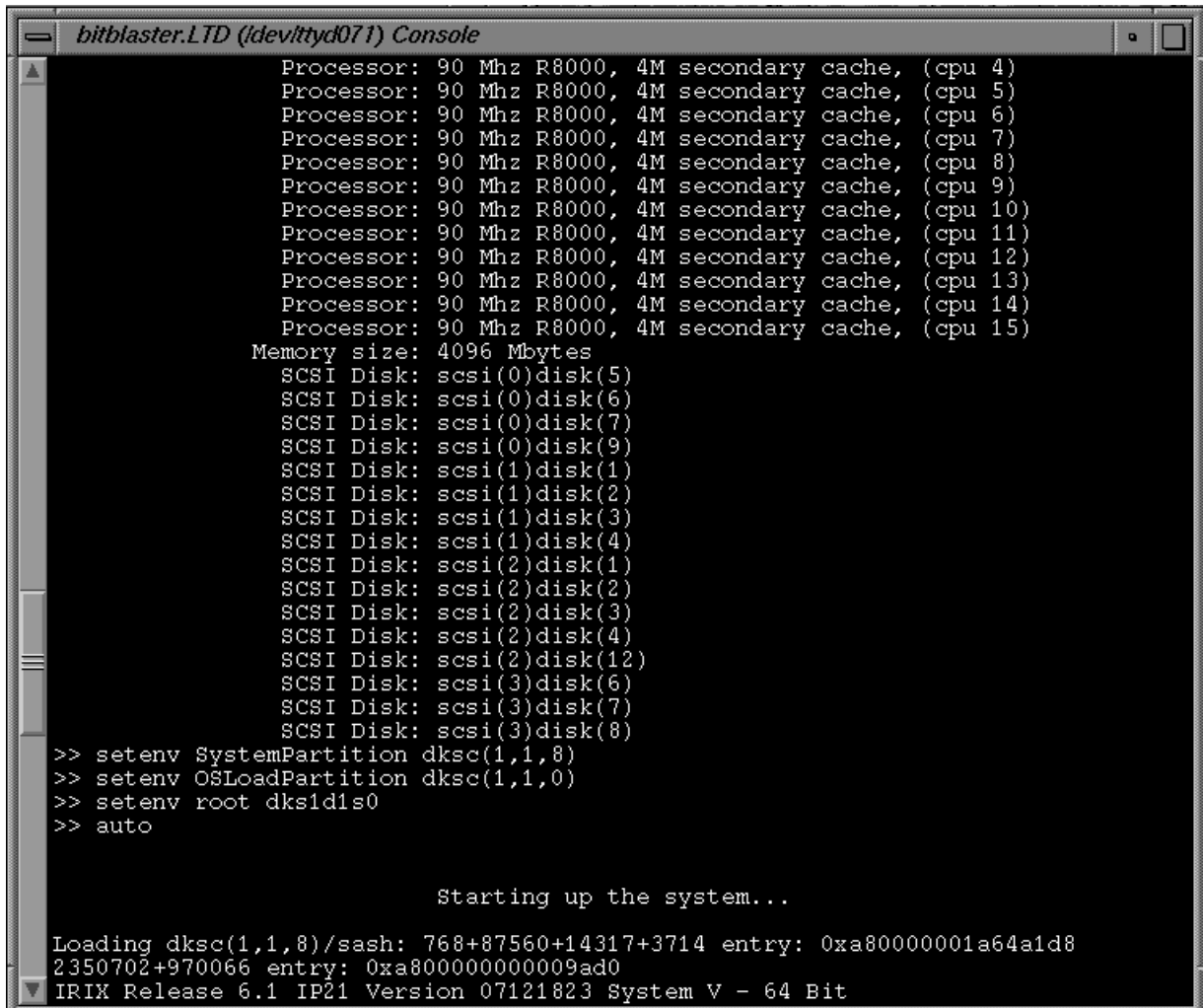


Figure 3-3 IRISconsole Hardware Status Window

The Get Console function opens an IRIX shell window that is the system console window for the selected system. Figure 3-4 shows a system console window in the middle of a reboot.



```
bitblaster.LTD (/dev/ttyd071) Console
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 4)
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 5)
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 6)
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 7)
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 8)
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 9)
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 10)
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 11)
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 12)
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 13)
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 14)
Processor: 90 Mhz R8000, 4M secondary cache, (cpu 15)
Memory size: 4096 Mbytes
SCSI Disk: scsi(0)disk(5)
SCSI Disk: scsi(0)disk(6)
SCSI Disk: scsi(0)disk(7)
SCSI Disk: scsi(0)disk(9)
SCSI Disk: scsi(1)disk(1)
SCSI Disk: scsi(1)disk(2)
SCSI Disk: scsi(1)disk(3)
SCSI Disk: scsi(1)disk(4)
SCSI Disk: scsi(2)disk(1)
SCSI Disk: scsi(2)disk(2)
SCSI Disk: scsi(2)disk(3)
SCSI Disk: scsi(2)disk(4)
SCSI Disk: scsi(2)disk(12)
SCSI Disk: scsi(3)disk(6)
SCSI Disk: scsi(3)disk(7)
SCSI Disk: scsi(3)disk(8)
>> setenv SystemPartition dksc(1,1,8)
>> setenv OSLoadPartition dksc(1,1,0)
>> setenv root dks1dis0
>> auto

Starting up the system...

Loading dksc(1,1,8)/sash: 768+87560+14317+3714 entry: 0xa80000001a64a1d8
2350702+970066 entry: 0xa800000000009ad0
IRIX Release 6.1 IP21 Version 07121823 System V - 64 Bit
```

Figure 3-4 IRISconsole System Console Window

Using Other Tools With IRISconsole

The same workstation that runs IRISconsole can be used simultaneously to run other administrative and diagnostic software such as Performance Co-pilot.

You or a system programmer can run Performance Co-Pilot in order to analyze the performance of the Array. Performance Co-Pilot works well with IRISconsole: from system console windows you can start and stop processes, and with PCP you can observe the performance effects.

Other products for monitoring and analysis can be executed at the console workstation so that all your display tools are on the same screen. For example, NetVisualyzer™ can be used to monitor network traffic concurrently with IRISconsole and Performance Co-Pilot.

You can also run the *aview* command at the console station, as another way of staying aware of the system status (see “Browsing With ArrayView” on page 40).

Table 3-1 lists information sources for the console management aids.

Table 3-1 Information Sources: Console Management

Topic	Book or URL	Book Number
IRISconsole	<i>IRISconsole Administrator's Guide</i> http://www.sgi.com/Products/hardware/challenge/IRISconsole.html	007-2872-xxx
IRISconsole hardware	<i>IRISconsole ST-1600 Installation Guide</i> <i>Indy Workstation Owner's Guide</i>	007-2839-xxx 007-9804-xxx
Performance Co-Pilot	<i>The Performance Co-Pilot User's and Administrator's Guide</i>	007-2614-xxx
NetVisualyzer	NetVisualyzer User's Guide	007-0812-xxx
<i>aview</i> command	<i>aview</i> (1) and “Browsing With ArrayView” on page 40	

About Array Configuration

The system administrator has to initialize the Array configuration database, a file that is used by the Array Services daemon in executing almost every *ainfo* and *array* command. For details about array configuration, see the reference pages cited in Table 3-2.

Table 3-2 Information Sources: Array Configuration

Topic	Book, Reference Page, or URL	Book Number
Array Services overview	array_services(5)	
Array Services user commands	ainfo(1), array(1)	
Array Services daemon overview	arrayd(1m)	
Configuration file format	arrayd.conf(4), <i>/usr/lib/array/arrayd.conf.template</i>	
Configuration file validator	ascheck(1)	
Array Services simple configurator	arrayconfig(1m)	

About the Uses of the Configuration File

The configuration files are read by the Array Services daemon when it starts. Normally it is started in each node during the system bootstrap. (You can also run the daemon from a command line in order to check the syntax of the configuration files.)

The configuration files inform the daemon of the data needed by *ainfo* and *array*:

- The names of Array systems, including the current Array but also any other Arrays on which a user could run an Array Services command (reported by *ainfo*).
- The names and types of the nodes in each named Array, especially the hostnames that would be used in an Array Services command (reported by *ainfo*).

- The authentication keys, if any, that must be used with Array Services commands (required as *-Kl* and *-Kr* command options, see “Summary of Common Command Options” on page 34).
- The commands that are valid with the *array* command.

About Configuration File Format and Contents

A configuration file is a readable text file. The file contains entries of the following four types, which are detailed in later topics.

Array definition	Describes this array and other known arrays, including array names and the node names and types.
Command definition	Specifies the usage and operation of a command that can be invoked through the <i>array</i> command.
Authentication	Specifies authentication numbers that must be used to access the Array.
Local option	Options that modify the operation of the other entries or <i>arrayd</i> .

Blank lines, white space, and comment lines beginning with “#” can be used freely for readability. Entries can be in any order in any of the files read by *arrayd*.

Besides punctuation, entries are formed with a keyword-based syntax. Keyword recognition is not case-sensitive; however keywords are shown in uppercase in this text and in the reference page. The entries are primarily formed from keywords, numbers, and quoted strings, as detailed in the reference page *arrayd.conf(4)*.

Loading Configuration Data

The Array Services daemon, *arrayd*, one or more filenames as arguments. It reads them all, and treats them like logical continuations (in effect, it concatenates them). If no filenames are specified, it reads */usr/lib/array/arrayd.conf* and */usr/lib/array/arrayd.auth*. A different set of files, and any other *arrayd* command-line options, can be written into the file */etc/config/arrayd.options*, which is read by the */etc/init.d/array* script that launches *arrayd* at boot time.

Since configuration data can be stored in two or more files, you can combine different strategies, for example:

- One file can have different access permissions than another. Typically, `/usr/lib/array/arrayd.conf` is world-readable and contains the available *array* commands, while `/usr/lib/array/arrayd.auth` is readable only by root and contains authentication codes.
- One node can have different configuration data than another. For example, certain commands might be defined only in certain nodes; or only the nodes used for interactive logins might know the names of all other nodes.
- You can use NFS-mounted configuration files. You could put a small configuration file on each machine to define the Array and authentication keys, but you could have a larger file defining *array* commands that is NFS-mounted from one node.

After you modify the configuration files, you can make *arrayd* reload them by killing the daemon and restarting it in each machine. The script `/etc/init.d/array` supports this operation: execute

```
/etc/init.d/array stop
```

to kill the daemon, and

```
/etc/init.d/array restart
```

to kill and restart it in one operation.

The Array Services daemon in any node knows only the information in the configuration files available in that node. This can be an advantage, in that you can limit the use of particular nodes; but it does require that you take pains to keep common information synchronized. (An automated way to do this is sketched under “Designing New Array Commands” on page 62.)

About Substitution Syntax

The reference page `arrayd.conf(4)` details the syntax rules for forming entries in the configuration files. An important feature of this syntax is the use of several kinds of text substitution, by which variable text is substituted into entries when they are executed.

Most of the supported substitutions are used in Command entries. These substitutions are performed dynamically, each time the *array* command invokes a subcommand. At that time, substitutions insert values that are unique to the invocation of that

subcommand. For example, the value `%USER` inserts the user ID of the user who is invoking the `array` command. Such a substitution has no meaning except during execution of a command.

Substitutions in other configuration entries are performed only once, at the time the configuration file is read by `arrayd`. Only environment variable substitution makes sense in these entries. The environment variable values that are substituted are the values inherited by `arrayd` from the script that invokes it, which is `/etc/init.d/array`.

Testing Configuration Changes

The configuration files contain many sections and options (detailed in the topics that follow this one). The Array Services command `ascheck` performs a basic sanity check of all configuration files in the Array.

After making a change, you can test an individual configuration file for correct syntax by executing `arrayd` as a command with the `-c` and `-f` options. For example, suppose you have just added a new command definition to `/usr/lib/array/arrayd.local`. You can check its syntax with the command

```
arrayd -c -f /usr/lib/array/arrayd.local
```

When testing new commands for correct operation, you need to see the warning and error messages produced by `arrayd` and processes that it may spawn. The `stderr` messages from a daemon are not normally visible. You can make them visible by the following procedure:

1. On one node, kill the daemon.
2. In one shell window on that node, start `arrayd` with the options `-n -v`. Instead of moving into the background, it remains attached to the shell terminal.

Note: Although `arrayd` becomes functional in this mode, it does not refer to `/etc/config/arrayd.options`, so you need to specify explicitly all command-line options, such as the names of nonstandard configuration files.

3. From another shell window on the same or other nodes, issue `ainfo` and `array` commands to test the new configuration data. Diagnostic output appears in the `arrayd` shell window.
4. Terminate `arrayd` and restart it as a daemon (without `-n`).

During steps 1, 2, and 4, the test node may fail to respond to `ainfo` and `array` commands, so users should be warned that the Array is in test mode.

Configuring Arrays and Machines

Each ARRAY entry gives the name and composition of an Array system that users can access. At least one ARRAY must be defined at every node, the Array in use.

Specifying Arrayname and Machine Names

A simple example of an ARRAY definition is as follows:

```
array simple
    machine congo
    machine niger
    machine nile
```

The arrayname *simple* is the value the user must specify in the *-a* option (see “Summary of Common Command Options” on page 34). One arrayname should be specified in a DESTINATION ARRAY local option as the default array (reported by *ainfo dflt*). Local options are listed under “Configuring Local Options” on page 62.

The MACHINE subentries of ARRAY define the nodenames that the user can specify with the *-s* option. These names are also reported by the command *ainfo machines*.

Specifying IP Addresses and Ports

The simple MACHINE subentries shown in the example are based on the assumption that the hostname is the same as the machine’s name to Domain Name Services (DNS). If a machine’s IP address cannot be obtained from the given hostname, you must provide a HOSTNAME subentry to specify either a completely-qualified domain name or an IP address, as follows

```
array simple
    machine congo
        hostname congo.engr.hitech.com
        port 8820
    machine niger
        hostname niger.engr.hitech.com
    machine nile
        hostname "198.206.32.85"
```

The preceding example also shows how the PORT subentry can be used to specify that *arrayd* in a particular machine uses a different socket number than the default 5434.

Specifying Additional Attributes

Under both ARRAY and MACHINE you can insert “attributes,” which are named string values. These attributes are not used by Array Services, but they are displayed by *ainfo* and can be returned to programs using the Array Services library (“Array Services Library” on page 76). Some examples of attributes would be as follows:

```
array simple
  array_attribute config_date="04/03/96"
  machine a_node
    machine_attribute aka="congo"
    hostname congo.engr.hitech.com
```

Tip: You can write code that fetches any arrayname, machine name, or attribute string from any node in the array. See “Database Interrogation” on page 80.

Configuring Authentication Codes

In Array 3.0 only one type of authentication is provided: a simple numeric key that can be required with any Array Services command. You can specify a single authentication code number for each node. The user must specify the code with any command entered at that node, or addressed to that node using the *-s* option (see “Summary of Common Command Options” on page 34).

The *arshell* command is like *rsh* in that it runs a command on another machine under the *userid* of the invoking user. Use of authentication codes makes Array Services somewhat more secure than *rsh*.

Configuring Array Commands

The user can invoke arbitrary IRIX commands on single nodes using the *arshell* command (see “Using arshell” on page 42). The user can also launch MPI and PVM programs that automatically distribute over multiple nodes. However, the only way to launch coordinated IRIX programs on all nodes at once is to use the *array* command. This command does not accept any IRIX command; it only permits execution of commands that the administrator has configured into the Array Services database.

You can define any set of commands that your users need. You have complete control over how any single Array node executes a command (the definition can be different in different nodes). A command can simply invoke a standard IRIX command, or, since you can define a command as invoking a script, you can make a command arbitrarily complex.

Operation of Array Commands

When a user invokes the *array* command, the subcommand and its arguments are processed by the destination node specified by *-s*. Unless the *-l* option was given, that daemon also distributes the subcommand and its arguments to all other array nodes that it knows about (the destination node might be configured with only a subset of nodes). At each node, *arrayd* searches the configuration database for a COMMAND entry with the same name as the array subcommand.

For example, when the user enters

```
array -s tokyo uptime
```

the subcommand *uptime* is processed by *arrayd* in node *tokyo*. When it finds the subcommand valid, it distributes it to every node that is configured in the default array at node *tokyo*.

The COMMAND entry for *uptime* is distributed in this form (you can read it in the file */usr/lib/array/arrayd.conf*).

```
command uptime          # Display uptime/load of all nodes in array
      invoke /usr/lib/array/auptime %LOCAL
```

The INVOKE subentry tells *arrayd* how to execute this command. In this case, it executes a shell script */usr/lib/array/auptime*, passing it one argument, the name of the local node. This command is executed at every node, with %LOCAL replaced by that node's name.

Summary of Command Definition Syntax

Look at the basic set of commands distributed with Array 3.0 (*/usr/lib/array/arrayd.conf*). Each COMMAND entry is defined using the subentries shown in Table 3-3. (These are described in great detail in reference page *arrayd.conf(4)*.)

Table 3-3 Subentries of a COMMAND Definition

Keyword	Meaning of Following Values
COMMAND	The name of the command as the user gives it to <i>array</i> .
INVOKE	An IRIX command to be executed on every node. The argument values can be literals, or arguments given by the user, or other substitution values.
MERGE	An IRIX command to be executed only on the distributing node, to gather the streams of output from all nodes and combine them into a single stream.
USER	The userid under which the INVOKE and MERGE commands run. Usually given as USER %USER, so as to run as the user who invoked <i>array</i> .
GROUP	The groupname under which the INVOKE and MERGE commands run. Usually given as GROUP %GROUP, so as to run in the group of the user who invoked <i>array</i> (see reference page groups(1)).
PROJECT	The project under which the INVOKE and MERGE commands run. Usually given as PROJECT %PROJECT, so as to run in the project of the user who invoked <i>array</i> (see reference page projects(5)).
OPTIONS	A variety of options to modify this command; see Table 3-5.

The IRIX commands called by INVOKE and MERGE must be specified as full pathnames, because *arrayd* has no defined execution path. As with a shell script, these IRIX commands are often composed from a few literal values and many substitution strings. The substitutions that are supported (which are documented in detail in the *arrayd.conf*(4) reference page) are summarized in Table 3-4.

Table 3-4 Substitutions Used in a COMMAND Definition

Substitution	Replacement Value
%1..%9; %ARG(<i>n</i>); %ALLARGS; %OPTARG(<i>n</i>)	Argument tokens from the user’s subcommand. %OPTARG does not produce an error message if the specified argument is omitted.
%USER, %GROUP, %PROJECT	The effective userid, effective groupid, and project of the user who invoked <i>array</i> .

Table 3-4 (continued) Substitutions Used in a COMMAND Definition

Substitution	Replacement Value
%REALUSER, %REALGROUP	The real userid and real groupid of the user who invoked <i>array</i> .
%ASH	The ASH under which the INVOKE or MERGE command is to run.
%PID(<i>ash</i>)	List of PID values for a specified ASH. %PID(%ASH) is a common use.
%ARRAY	The arrayname, either default or as given in the <i>-a</i> option.
%LOCAL	The hostname of the executing node.
%ORIGIN	The full domain name of the node where the <i>array</i> command ran and the output is to be viewed.
%OUTFILE	List of names of temporary files, each containing the output from one node's INVOKE command (valid only in the MERGE subentry).

The OPTIONS subentry permits a number of important modifications of the command execution; these are summarized in Table 3-5.

Table 3-5 Options of the COMMAND Definition

Keyword	Effect on Command
LOCAL	Do not distribute to other nodes (effectively forces the <i>-l</i> option).
NEWSSESSION	Execute the INVOKE command under a newly-minted ASH. %ASH in the INVOKE line is the new ASH. The MERGE command runs under the original ASH, and %ASH substitutes as the old ASH in that line.
SETRUID	Set both the real and effective user ID from the USER subentry (normally USER only sets the effective UID).
SETRGID	Set both the real and effective group ID from the GROUP subentry (normally GROUP sets only the effective GID).
QUIET	Discard output of INVOKE, unless if MERGE subentry is given, pass INVOKE output to MERGE as usual and discard the MERGE output.
NOWAIT	Discard output and return as soon as the processes are invoked; do not wait for completion (a MERGE subentry is ineffective).

Configuring Local Options

The LOCAL entry specifies options to *arrayd* itself. The most important options are summarized in Table 3-6.

Table 3-6 Subentries of the LOCAL Entry

Subentry	Purpose
DIR	Pathname for the <i>arrayd</i> working directory, which is the initial, current working directory of INVOKE and MERGE commands. The default is <i>/usr/lib/array</i> .
DESTINATION ARRAY	Name of the default array, used when the user omits the -a option. When only one ARRAY entry is given, it is the default destination.
USER, GROUP, PROJECT	Default values for COMMAND execution when USER, GROUP, or PROJECT are omitted from the COMMAND definition.
HOSTNAME	Value returned in this node by %LOCAL. Default is the hostname.
PORT	Socket to be used by <i>arrayd</i> .

If you do not supply LOCAL USER, GROUP, and PROJECT values, the default values for USER and GROUP are “guest.”

Note: The HOSTNAME entry is needed whenever the IRIX *hostname* command does not return a node name as specified in the ARRAY MACHINE entry. In order to supply a LOCAL HOSTNAME entry unique to each node, each node needs an individualized copy of at least one configuration file.

Designing New Array Commands

A basic set of commands is distributed in the file */usr/lib/array/arrayd.conf.template*. You should examine this file carefully before defining commands of your own. You can define new commands which then become available to the users of the Array system.

Typically, a new command will be defined with an INVOKE subentry that names a script written in *sh*, *csh*, or *perl* syntax. You use the substitution values to set up arguments to the script. You use the USER, GROUP, PROJECT, and OPTIONS subentries to establish the execution conditions of the script. For one example of a command definition using a simple script, see “About the Distributed Example” on page 43.

Within the invoked script you can write any amount of logic to verify and validate the arguments, and to execute any sequence of commands. For an example of a script in *perl*, see */usr/lib/array/aps*, which is invoked by the *array ps* command.

Tip: *perl* is a particularly interesting choice for *array* commands, since *perl* has native support for socket I/O. In principle at least, you could build a distributed application in *perl* in which multiple instances are launched by *array* and coordinate and exchange data using sockets. Performance would not rival the highly tuned MPI and PVM libraries, but development would be simpler.

The administrator has need for distributed applications as well, since the configuration files are distributed over the Array. Here is an example of a distributed command to reinitialize the Array Services database on all nodes at once. The script to be executed at each node, called */usr/lib/array/arrayd-reinit* would read as follows:

```
#!/bin/sh
# Script to reinitialize arrayd with a new configuration file
# Usage:  arrayd-reinit <hostname:new-config-file>
sleep 10      # Let old arrayd finish distributing
rcp $1 /usr/lib/array/
/etc/init.d/array restart
exit 0
```

The script uses *rcp* to copy a specified file (presumably a configuration file such as *arrayd.conf*) into */usr/lib/array* (this will fail if %USER is not privileged). Then the script restarts *arrayd* (see */etc/init.d/array*) to reread configuration files.

The command definition would be as follows:

```
command reinit
    invoke /usr/lib/array/arrayd-reinit %ORIGIN:%1
    user  %USER
    group %GROUP
    options nowait    # Exit before restart occurs!
```

The INVOKE subentry calls the restart script shown above. The NOWAIT option prevents the daemon's waiting for the script to finish, since the script will kill the daemon.

Performance-Driven Programming in Array 3.0

Array 3.0 offers developers a rich set of compilers, libraries, system services, and development tools. Most of these facilities are documented separately. This chapter surveys the development tools and provides pointers to their documentation, as well as taking a deeper look at the Array Services library functions. The main topics include:

- “Basic Array Application Tuning Strategy” on page 66 includes some advice on performance tuning.
- “Locality, Latency, and Bandwidth” on page 73 discusses performance values for MPI and TCP/IP.
- “Array Services Library” on page 76 details the use of the Array Services functions.

Basic Array Application Tuning Strategy

Quite often, new applications developed for an Array system run with satisfactory job execution time. When execution times are not satisfactory, the developer must tune the application for improved performance.

An efficient, systematic tuning strategy helps you achieve the best possible performance in the minimum development time. One such tuning strategy is outlined in this section.

Tuning Single-Node Performance

The first step in tuning a parallel application—and it is a large step—is to tune it for best performance on a single node. Tuning any program begins with instrumenting the program to identify the parts where it spends excess time. Analyze these parts to determine whether algorithmic changes are possible. Optimizations of program logic and algorithms, when they are possible, always yield the largest improvements at the lowest cost.

When you are sure the algorithm is optimal and its coding is logically correct, examine the program for library use, cache use, software pipelining, and SMP performance.

Library Selection

There are many numerical libraries available, some from Silicon Graphics, Inc. and some from other sources both commercial and public domain. Every implementation of a standard algorithm has different characteristics for accuracy bounds and error propagation, raw speed, and speed as a ratio of the problem set size.

SGI Cray Scientific Library (SCSL) includes algorithms that are carefully coded and optimized to Silicon Graphics, Inc. hardware. The SCSL supercedes the older CHALLENGEcomplib™ product. But by all means have a selection of libraries available and try them all. Some library sources are listed in Table 4-1.

Table 4-1 Information Sources: General Numerical Libraries

Topic	Book or URL	Book Number
Directory of WWW software sources	http://www.yahoo.com/Science/Mathematics/Software/	
Vast collection of numerical software	http://www.netlib.org/	
Index to math and statistical software	http://gams.nist.gov/	
Visualization tools for physics	http://www.lassp.cornell.edu/LASSPTools/LASSPTools.html	
Volume Renderer and other tools	http://www.arc.umn.edu/gvl-software/gvl-software.html	

Some library collections (including CHALLENGEcomplib) are specifically tuned for parallel execution. Pointers to some sources of parallel libraries are listed in Table 4-2.

Table 4-2 Information Sources: Libraries for Parallel Computation

Topic	Book, Reference Page, or URL	Book Number
CHALLENGEcomplib overview	http://www.sgi.com/Products/Challengecomplib.html	
Center for Research in Parallel Computation	http://www.crpc.rice.edu/	
HPPC software collection	http://www.netlib.org/nhse/home.html	
HPCC Vendors "Mall"	http://www.npac.syr.edu/infomall/	

Tuning Information

Information sources about performance tuning and the Workshop tools are listed in Table 4-3.

Table 4-3 Information Sources: Performance Analysis Tools

Topic	Book, Reference Page, or URL	Book Number
Developer Magic overviews	<i>Developer Magic: ProDev WorkShop Overview</i> http://www.sgi.com/Products/WorkShop.html	007-2582-xxx
Debugger	<i>Developer Magic: Debugger User's Guide</i>	007-2579-xxx
Performance Analysis	<i>Developer Magic: Performance Analyzer User's Guide</i>	007-2581-xxx
Performance Analysis	<i>Developer Magic: Static Analyzer User's Guide</i>	007-2580-xxx
Performance Analysis (command-line tools)	<i>MIPSpro Compiling and Performance Tuning Guide</i>	007-2360-xxx
Performance Tuning on Origin2000 and Onyx2	<i>Performance Tuning Optimization for Origin2000 and Onyx2</i> , online only at http://www.sgi.com/techpubs/lib/makepage.cgi?007-3430-001	007-3430-xxx

Pipelining

The MIPSpro compilers sold by Silicon Graphics, Inc. support software pipelining, in which the compiler structures the machine code of compute-intensive loops to optimally schedule operations through the R8000 or R10000 CPU. The proper use of software pipelining can make immense differences in the speed of execution of certain loops. However, you must sometimes adjust the source code of a loop in order for the compiler to recognize it as eligible for pipeline treatment.

Software pipelining is discussed in the books shown in Table 4-4.

Table 4-4 Information Sources: Software Pipelining

Topic	Book, Reference Page, or URL	Book Number
Software pipelining	<i>MIPSpro 64-Bit Porting and Transition Guide</i>	007-2391-xxx
Software pipelining	<i>MIPSpro Compiling and Performance Tuning Guide</i>	007-2360-xxx

SMP Performance

You can make a computation-intensive program faster by applying multiple CPUs in parallel, within a single node of an array (within any Challenge-class server). You should explore single-node parallelism carefully before you even consider multinode parallel execution. The reasons are, first, the extensive support for parallel execution provided by software such as Power Fortran (77 and 90) and IRIS Power C; second, the relative ease of starting, running, and testing a program within one node; and finally the fact that a multinode program must be written to use the more complex model provided by High-Performance Fortran (HPF), by MPI, or by PVM.

If you are sure that a program will be a multinode program, or if you are working on a program that is already written to use MPI or PVM, you can still consider some parallel execution within each node. The parallelizing directives of Power Fortran or Power C can be applied in the context of a single source module. For example, you might use MPI to distribute an array in 1 MB sections to each node, but use single-node parallelism in the DO-loop that processes one section.

Parallel Performance Goals

When a program runs efficiently on a single node, but you find you still need to recruit more CPU cycles to it, you can look for further performance gains through multinode parallelism.

Be aware first that multiparallelism is far from a panacea. It is important to consider Amdahl's Law. If the code that can be run in parallel consumes less than 95% of the total execution time, targeting fewer than 18 processors is sufficient to realize any potential benefit from parallelization. When this is the case, parallelism within a single node is the most appropriate strategy (presuming at least one node has sufficient CPUs installed).

When the parallelization potential is above 96%, then parallelizing across the nodes of the Array can result in a performance speedup.

Designing Appropriate Parallel Algorithms

When designing a multinode application, your basic strategy should be to maximize the work done on the data within any node before communication between nodes is required. In general, aim to communicate between nodes as rarely as possible, and when communication is needed, to use the largest message units possible. This strategy maximizes the use of the high bandwidth and lower latencies of the bus within a node, as compared to the relatively slower communications of the HIPPI network.

While the implementation of your internode parallel algorithms may use a shared memory model (HPF) or message-passing model (MPI, PVM), the design of the fundamental algorithms is conceptually similar to the design of good parallel algorithms for a single-node program. To achieve the performance advantages of distributed multiprocessing, you must

- Partition the application into concurrent processes
- Implement efficient communication between the parallel processes to synchronize and exchange data
- Balance the workload among the parallel processes
- Minimize communication overhead, so that processors are well utilized

Test and Debug on a Single-node Server

If you design your applications to accept variable numbers of processors and problem sizes, often you can debug your fundamental program logic within a single node. When executing within one node you can apply all the debugging and visualization tools of the Workshop suite.

Parallel Programming and Communication Paradigms

Several models of parallel computation are available for the IRIX and Array 3.0. These models are discussed and compared in detail in the book listed in Table 4-5.

Table 4-5 Information Sources: Parallel Computation Models

Topic	Book, Reference Page, or URL	Book Number
Models of Parallel Computation	<i>Topics In IRIX Programming</i>	007-2478-xxx

The models are summarized here, but read the chapter of the book shown for details and the latest version information.

Shared-Memory Communication

IRIX provides several facilities that permit parallel processes to communicate by directly reading and writing the same address space.

Shared Memory within One Node

Three different interfaces provide shared-memory facilities within a single node.

- IRIX native shared memory
- POSIX 1003.1b shared memory (available as a patch to IRIX 6.2)
- System V Release 4 compatible shared memory

A variety of coordination primitives are available for synchronization and mutual exclusion within nodes:

- IRIX native mutual exclusion locks, semaphores, and barriers
- POSIX 1003.1b semaphores (available as a patch to IRIX 6.2)
- System V Release 4 compatible semaphores

You can call on these facilities directly in C programs. The POWER C and POWER Fortran runtime modules for parallel execution use the IRIX native shared memory to communicate, since IRIX shared-memory support is closely integrated with IRIX lightweight processes.

Shared Memory Between Nodes

The current Array architecture restricts direct shared memory IPC to only intranode communications. High Performance Fortran (HPF) provides a form of indirect shared memory for internode communications.

Message-Passing IPC

The message-passing communication model provides a “mail delivery” paradigm for interprocess communication. A collection of data items is given an identifier and “mailed” to a destination process, which subsequently receives it.

The message-passing model makes a clean separation between program modules, affording some protection from accidental changes to shared memory. However, any message-passing facility must incur some delay compared to shared memory, due to buffering, abstraction, and (sometimes) copying and transmission overheads.

Message Passing within a Node

In a C program, you can call on either of two interfaces for queue-based message passing within one node:

- System V Release 4 message queues
- POSIX 1003.1b message queues

The POSIX implementation (available as a patch for IRIX 6.2) uses shared memory and operates primarily in user space for minimal overhead. The SVR4 library is included primarily for compatibility, and incurs the overhead of a kernel calls.

Distributed Message Passing

Three abstract models are supported to allow the exchange of arbitrary messages between processes operating in the same or different nodes of an array:

- MPI (Message Passing Interface) is the preferred message-passing facility for Silicon Graphics, Inc. Array systems. The version of MPI distributed with Array 3.0 is carefully tuned to take maximum advantage of the HIPPI interconnect, and of shared memory within a node.
- PVM (Portable Virtual Machine) is supported for compatibility, and has been tuned to some extent to work correctly in an Array system.
- IRIX contains standard support for sockets, with which you can write programs that communicate between any two nodes on the internet. Array Services uses sockets to pass commands and messages between nodes (see "Using Array Services Commands" on page 33).

Hybrid Models

If you are preparing a new application, Silicon Graphics, Inc. recommends that you plan as follows:

- For implicit parallelism within a node, use the compiler facilities of the MIPSpro Fortran and C compilers, aided by Power Fortran and IRIS Power C.

- For explicit communication within a node, use either IRIX native shared memory, POSIX shared memory, or POSIX message queues.
- For distributed parallel execution, use MPI.

There is no requirement that applications use any one set of facilities exclusively. For example, the following common models are possible, among others:

- Shared-memory program with n processes in one node.
- Message-passing program with n processes in one node.
- Hybrid application with n processes in one node, using a combination of message passing and shared memory.
- Message-passing program with n processes distributed over p nodes, $n > p$.
- Hybrid application with n processes over p nodes, communicating between nodes via MPI but using shared memory to coordinate multiple processes within each node.

However, when designing a program to use a hybrid model, you must be aware that the MPI library is not “thread-safe,” that is, it has global variables with values that can be destroyed if it is executed by two lightweight processes concurrently. The MPI library should be entered by only one process in any share group. This is discussed in more detail in the *MPI and PVM User’s Guide*, 007-3286-xxx.

Locality, Latency, and Bandwidth

All forms of interprocess communication incur some delay. The time $t(s)$ required to communicate a message containing s bytes of data to another process can be roughly separated into a fixed overhead latency L that is independent of message size, and a size-dependent overhead, which represents the message size divided by the communication bandwidth B . The following formula is often used to approximate the time to transmit a message of length s :

$$t(s) = L + \frac{1}{B} \cdot s$$

MPI Communication Delays

Array 3.0 contains an optimized protocol stack supporting MPI protocols on HIPPI. Separate design approaches have been implemented for short messages and long ones. Special attention is paid to latency for shorter messages, which are more common. The advice given in “Reducing the Effect of Communication Delay” on page 74, to use fewer, longer messages, is valid, but the reduced latency from prior versions should improve the performance of many programs.

Other conditions can affect the use of HIPPI by MPI. When four or more applications are contending for the use of an adapter, MPI does not use that adapter. (The limit is 8 applications per adapter on an Origin2000 node). When a node does not have a HIPPI adapter, or when the maximum MPI jobs are contending for all available adapters, an internode MPI transfer uses a socket instead. This case is not optimized and will be slower.

TCP/IP Communication Delays

The basic IRIX support for TCP/IP is not changed for Array 3.0, and does not take advantage of the special MPI tuning. As a result, you are strongly advised to construct a distributed application using MPI, not sockets.

When you must use sockets, be aware that the bandwidth you can achieve can vary over an extremely wide range depending on several factors. The most important are: the size of the socket buffer (the `SO_SNDBUF` and `SO_RCVBUF` options of `setsockopt(0)`) and the size of the message.

Typical performance with a 62 KB socket buffer and a stream of 16 KB messages is approximately 15 MB/second. Much higher speeds, up to 60 MB/sec and more, can be achieved using larger, page-aligned transfers, with buffer pages locked in memory. Rates of 90 MB/sec can be achieved by highly-tuned benchmark programs.

Reducing the Effect of Communication Delay

To effectively exploit the memory hierarchy of the Array, you must be aware of the effects of communication latency and bandwidth while designing your program. Your best strategy is to send fewer, longer messages, and to overlap communication with useful work when possible.

Do Not Use Message Aggregation

You can sometimes reduce message count by aggregating small messages to the same destination into one larger message. You may find an existing MPI program going to some effort to block, or aggregate, messages. This is no longer recommended.

The HIPPI support in Array 3.0 is designed to incur low latency for small messages. The extra program logic needed to block and unblock messages will likely cost as much time as it saves. It is true in general that the fewer the messages, the better; but with Array 3.0 you should simply send a small message as soon as it is ready.

Overlap Processing with Communication

Try to design the application to overlap message delays with computation. MPI permits a process to send messages asynchronously. The `MPI_Isend()` function returns from the immediately, before the message has been sent. MPI also permits asynchronous receipt; the `MPI_Irecv()` function tests for available data without waiting when none is ready.

Asynchronous communication permits the sender or receiver to continue computation while data is transferred by the system. Figure 4-1 shows the potential performance advantages of asynchronous communication. Figure 4-1a is a time line of two standard communicating processes. Figure 4-1b is a time line of the same two processes using asynchronous sends and receives. Here, much of the communication time is hidden behind computation.

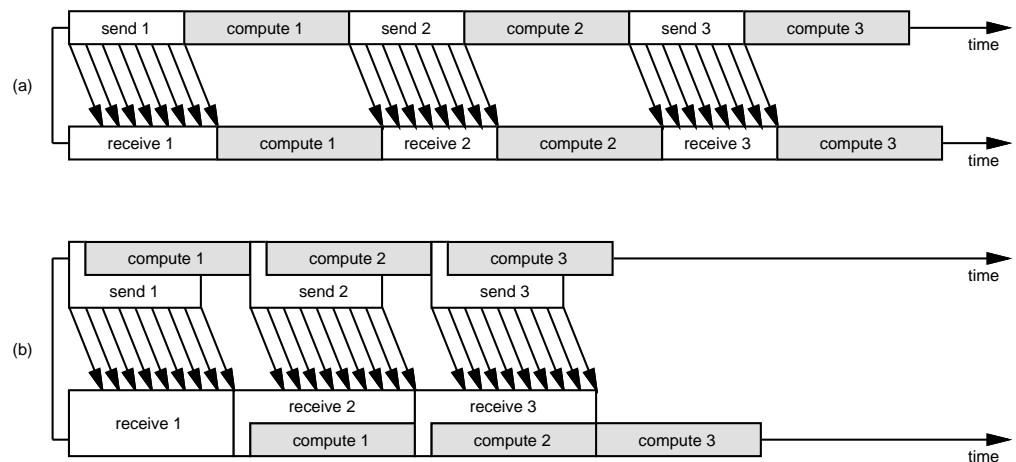


Figure 4-1 Gaining Efficiency Through Asynchronous Communication

When implementing asynchronous communication in MPI, it is important to use both **MPI_Isend()** and **MPI_Irecv()**. Neither **MPI_Send()** nor **MPI_Recv()** can begin to transfer data until a matching receive has been posted. Using **MPI_Irecv()** allows your program to post a receive earlier rather than later.

Asynchronous communication can in practice hide most communication delays, but it can require program restructuring. Optimize an existing program in other ways first; possibly the increased complexity in program logic will not be necessary.

Array Services Library

Array Services consists of a configuration database, a daemon (*arrayd*) that runs in each node to provide services, and several user-level commands. The facilities of Array Services are also available to developers through the Array Services library, a set of functions through which you can interrogate the configuration database and call on the services of *arrayd*.

The commands of Array Services are covered in “Using Array Services Commands” on page 33. The administration of Array Services is described in “About Array Configuration” on page 53 and topics that follow it. These topics are useful background information for understanding the Array Services library.

Array Services Library Overview

The programming interface to Array Services is declared in the header file */usr/include/arraysvcs.h*. The object code is located in */usr/lib/libarray.so*, included in a program by specifying *-larray* during compilation. The library is distributed in o32, n32, and 64-bit versions (not all need to be installed). The functions are documented in reference pages in volume 3.

The library functions can be grouped into these categories:

- Functions to connect to Array Services daemons in the local or other nodes, and to get and set *arrayd* options.
- Functions to interrogate the Array Services configuration database, listing arrays, nodes, and attributes of arrays and nodes.
- Functions to allocate Array Session Handles (ASHs), to query active ASHs and to change the relationship between PIDs and ASHs.

- A function to execute a command as for the *array* command (see “Operation of Array Commands” on page 59).
- A function to execute any arbitrary user command on an array node.

These functions are examined in following topics.

Data Structures

The Array Services functions work with a number of data structures that are declared in *arraysvcs.h*. In general, each data structure is allocated by one particular function, which returns a pointer to the structure as the function’s result. Your code uses the returned structure, possibly passing it as an argument to other functions.

When your code is finished with a structure, it is expected to call a specific function that frees that type of structure. If your code does not free each structure, a memory leak results.

The data structures and their contents are summarized in Table 4-6.

Table 4-6 Array Services Data Structures

Structure	Contents	FreedException
<i>asarray_t</i>	Name and attributes of an Array.	asfreearray()
<i>asarraylist_t</i>	List of <i>asarray_t</i> structures.	asfreearraylist()
<i>asashlist_t</i>	List of ASH values.	asfreeashlist()
<i>ascmdrslt_t</i>	Describes output of executing an <i>array</i> command on one node, including temporary files and socket numbers.	freed as part of a list
<i>ascmdrsltlist_t</i>	List of command results, one <i>ascmdrslt_t</i> per node where an <i>array</i> command was executed.	asfreecmdrsltlist()
<i>asmachine_t</i>	Configuration data about one node: machine name and attributes.	freed as part of a list
<i>asmachinelist_t</i>	List of <i>asmachine_t</i> structures, one per machine in the queried array	asfreemachinelist()
<i>aspidlist_t</i>	List of PID values.	asfreepidlist()

Error Message Conventions

The functions of the Array Services library have a complicated convention for error return codes. The reference pages related to this convention are listed in Table 4-7.

Table 4-7 Error Message Functions

Function	Operation
aserrorcode(3X)	Discusses the error code conventions and some macro functions used to extract subfields from an error code.
asmakeerror(3X)	Constructs an error code value from its component parts.
asstrerror(3X)	Returns a descriptive string for a given error code value.
aspperror(3X)	Prints a descriptive string, with a specified heading string, on stderr.

In general, each function sets a value in the global *aserrorcode*, which has type *aserror_t* (not necessarily an *int*). An error code is a structured value with these parts:

- *aserrno* is a general error number similar to those declared in *sys/errno.h*.
- *aserrwhy* documents the cause of the error.
- *aserrwhat* documents the component that detected the error.
- *aserrextra* may give additional information.

Macro functions to extract these subfields from the global *aserrorcode* are provided.

Connecting to Array Services Daemons

The functions listed in Table 4-8 are used to open a connection between the node where your program runs and an instance of *arrayd* in the same or another node.

Table 4-8 Functions for Connections to Array Services Daemons

Function	Operation
asopenserver(3X)	Establishes a logical connection to <i>arrayd</i> in a specified node, returning a token that represents that connection for use in other functions.
ascloseserver(3X)	Close an <i>arrayd</i> connection created by asopenserver() .

Table 4-8 (continued) Functions for Connections to Array Services Daemons

Function	Operation
asgetserveropt(3X)	Return the local options currently in use by an instance of <i>arrayd</i> .
asfltserveropt(3X)	Return the default options in effect at an instance of <i>arrayd</i> .
assetserveropt(3X)	Set new options for an instance of <i>arrayd</i> .

The key function is **asopenserver()**. It takes a nodename as a character string (as a user would give it in the *-s* option; see “Summary of Common Command Options” on page 34), and optionally a socket number to override the default *arrayd* socket number. This function opens a socket connection to the specified instance of *arrayd*. The returned token (type *asserver_t*) stands for that connection and is passed to other functions.

The functions for getting and setting server options can change the configured options shown in Table 4-9. To set these options is the programmatic equivalent of passing command line options in an Array Services command (see “About Array Configuration” on page 53 and “Summary of Common Command Options” on page 34).

Table 4-9 Server Options Functions Can Query or Change

Constant	Changeable?	Meaning
AS_SO_TIMEOUT	yes	Timeout interval for any request to this server.
AS_SO_CTIMEOUT	yes	Timeout interval for connecting to this server.
AS_SO_FORWARD	yes	Whether or not Array Services requests should be forwarded through the local <i>arrayd</i> , or sent directly (the <i>-F</i> option).
AS_SO_LOCALKEY	yes	The local authentication key (the <i>-Kl</i> command option).
AS_SO_REMOTEKEY	yes	The remote authentication key (<i>-Kr</i> command option).
AS_SO_PORTNUM	no	In default options only, the default socket number.
AS_SO_HOSTNAME	no	The hostname for this connection.

Database Interrogation

The functions summarized in Table 4-10 are used to interrogate the configuration database used by *arrayd* in a specified node (see “About Array Configuration” on page 53).

Table 4-10 Functions for Interrogating the Configuration

Function	Operation
asetdfltarray(3X)	Return the array name and all attributes strings for the default array known to a specified server, in an <i>asarray_t</i> structure.
aslistarrays(3X)	Return the names of all arrays, with their attribute strings, from a specified server, as an <i>asarraylist_t</i> structure.
aslistmachines(3X)	Return the names of all machines, with their attribute strings, from a specified server, as an <i>asmachinelist_t</i> structure.
asetattr(3X)	Search for a particular attribute name in a list of attribute strings, and return its value.

Using these functions you can extract any arrayname, nodename, or attribute that is known to an *arrayd* instance you have opened.

Managing Array Service Handles

The functions summarized in Table 4-11 are used to create and interrogate ASH values.

Table 4-11 Functions for Managing Array Service Handles

Function	Operation
asallocash(3X)	Allocate a new ASH value. The value is only created, it is not applied to any process.
aspidsinash(3X)	Returns a list of PID values associated with an ASH at a specified server, as an <i>aspidlist_t</i> structure.
asashofpid(3X)	Returns the ASH associated with a specified PID.
setash(2)	Change the ASH of the calling process.

The **asallocash()** function is like the command *ainfo newash* (see “About Array Session Handles (ASH)” on page 41). Only a program with root privilege can use the **setash()** system function to change the ASH of the current process. Unprivileged processes can create new ASH values but cannot change their ASH.

The functions summarized in Table 4-12 are used to enumerate the active ASH values at a specified node. In each case, the list of ASH values is returned in an *asashlist_t* structure.

Table 4-12 Functions for ASH Interrogation

Function	Operation
aslistashs(3X)	Return active ASH values from one node or all nodes of a specified Array via a specified server.
aslistashs_array(3X)	Return active ASH values from an Array by name.
aslistashs_server(3X)	Return active ASH values known to a specified server node.
aslistashs_local(3X)	Return active ASH values in the local node.
asashisglobal(3X)	Test to see if an ASH is global.

Executing an array Command

The **ascommand()** function is the programmatic equivalent of the *array* command (see “Operation of Array Commands” on page 59 and the *array(1)* reference page). This command has many options and can be used to execute commands in three distinct modes.

The command to be executed must be prepared in an *ascmdreq_t* structure, which contains the following fields:

```
typedef struct ascmdreq {
    char *array;           /* Name of target array */
    int flags;            /* Option flags */
    int numargs;         /* Number of arguments */
    char **args;         /* Cmd arguments (ala argv) */
    int ioflags;        /* I/O flags for interactive commands */
    char rsvrd[100];     /* reserved for expansion: init to 0's */
} ascmdreq_t;
```

Your program must prepare this structure in order to execute a command. The option flags allow for the same controls as the command line options of *array*.

The result of the command is returned as an *ascmdrsltlist_t* structure, which is a vector of *ascmdrslt_t* structures, one for each node at which the command was executed. Each *ascmdrslt_t* contains the following fields:

```
typedef struct ascmdrslt {
    char      *machine; /* Name of responding machine */
    ash_t     ash;      /* ASH of running command */
    int       flags;    /* Result flags */
    aerror_t  error;    /* Error code for this command */
    int       status;   /* Exit status */
    char      *outfile; /* Name of output file */
    int       ioflags;  /* I/O connections (see ascmdreq_t) */
    int       stdinfd;  /* File descriptor for command's stdin */
    int       stdoutfd; /* File descriptor for command's stdout */
    int       stderrfd; /* File descriptor for command's stderr */
    int       signalfd; /* File descriptor for sending signals */
} ascmdrslt_t;
```

The fields *machine*, *ash*, *flags*, *error*, and *status* reflect the result of the command execution in that machine. The other fields depend on the mode of execution.

Normal Batch Execution

To execute a command in the normal way, waiting for it to complete and collecting its output, you do not set either `ASCMDREQ_NOWAIT` or `ASCMDREQ_INTERACTIVE` in the command option flags.

Control returns from `ascommand()` when the command is complete on all nodes. If the `ASCMDREQ_OUTPUT` flag was specified, and if the command definition does not specify a `MERGE` subentry (see “Summary of Command Definition Syntax” on page 59), the *outfile* result field contains the name of a temporary file containing one node’s output stream.

When the command is implemented with a `MERGE` subentry, there is only one output file no matter how many nodes are invoked. In this case, the returned list contains only one *ascmdrslt_t* structure. It contains the `ASCMDRSLT_MERGED` and `ASCMDREQ_OUTPUT` flags, and the *outfile* result field contains the name of a temporary file containing the merged output.

Immediate Execution

When a command has no useful output and should execute concurrently with the calling program, you specify the `ASCMDREQ_NOWAIT` option. In this case, output cannot be collected because no program will be waiting to use it. Control returns as soon as the command has been distributed. The result structures do not reflect the command's result but only the result of trying to start it.

Interactive Execution

You can start a command in such a way that your program has direct interaction with the input and output streams of the command process in every node. When you do this, your program can supply input and inspect output in near real time.

To establish interactive execution, specify `ASCMDREQ_INTERACTIVE` in the command option flag. Also set one or more of the following flags in the *ioflags* field:

<code>ASCMDIO_STDIN</code>	Requests a socket attached to the command's stdin.
<code>ASCMDIO_STDOUT</code>	Requests a socket attached to the command's stdout.
<code>ASCMDIO_STDERR</code>	Requests a socket attached to the command's stderr.
<code>ASCMDIO_SIGNAL</code>	Requests a socket that can be used to deliver signals.

As with `ASCMDREQ_NOWAIT`, control returns as soon as the command has been distributed. Each result structure contains file descriptors for the requested sockets for the command process in that node.

Your program writes data into the *stdinfd* file descriptor of one node in order to send data to the stdin stream in that node. Your program reads data from the *stdoutfd* file descriptor to read one node's output stream.

You will typically use either the `select()` or the `poll()` system function to learn when one of the sockets is ready for use. You may choose to start one or more subprocesses using `fork()` to handle I/O to the sockets of each node (see the `select(2)`, `poll(2)` and `sproc(2)` reference pages). (You may also use `sproc()` to make subprocesses, but keep in mind that the `libarray` is not thread-safe, so it should only be used from one process in a share group.)

Executing a User Command

The **asrcmd()** function allows a program to initiate any user command string on a specified node. This provides a powerful facility for remote execution that does not require root privilege, as the standard **rcmd()** function does (compare the **asrcmd(3)** and **rcmd(3)** reference pages).

The **asrcmd()** function takes arguments specifying:

- The array node to use, as returned by **asopensever()** (see “Connecting to Array Services Daemons” on page 78).
- The user name to use on the remote node.
- The command line to be executed.

The returned value (as with **rcmd()**) is a socket that represents the standard input and output streams of the executing command. Optionally, a separate socket for the standard error stream can be obtained.

The RendAsunder Demo Program

This appendix describes the *RendAsunder* demonstration program that is supplied with Array 3.0 software.

RendAsunder is an interactive parallel software volume renderer. During the process of volume rendering, a three-dimensional array of data elements (each assigned a corresponding color and transparency) is rendered to a two-dimensional image, from a user-defined viewpoint.

A 375 MB volumetric data set from the National Library of Medicine's Visible Human Project is provided with RendAsunder. The data set consists of a 584 x 1878 x 341 element volume of eight-bit samples of a cryosectioned human male.

Note: Use of this demonstration implies agreement with the terms of the contract described in the file */usr/array/gifts/RendAsunder/Readme*. Read this file before executing the demonstration.

Starting RendAsunder

Install the component *RendAsunder.sw.base* from the array CD on any node of the array. Execute `/usr/sbin/RendAsunder` on that same node to begin execution of *RendAsunder* on the entire array.

Setting Up the Configuration File

RendAsunder keeps a configuration file of its own to describe the array. When you launch `/usr/sbin/RendAsunder`, it uses Array Services to generate a configuration file in `/usr/array/gifts/RendAsunder/config/curr.config`. It then runs the executable `/usr/array/gifts/RendAsunder/start` to launch the demo.

Once you have started the program this way, you can later use `/usr/array/gifts/RendAsunder/start` directly to start without regenerating the configuration.

Setting the Graphics Display

The graphics appear as determined by the standard X Windows environment variable `DISPLAY`. For best results, output should be displayed on a graphics device directly attached to a node. `DISPLAY` should be set to select a graphics-equipped node, for example

```
setenv DISPLAY bitblaster:0
```

If no node is graphics-equipped, you can display graphics on a remote host, although at some cost in performance. To do so, set `DISPLAY` to select the remote host.

The Graphics Window

When started, *RendAsunder* opens a control panel and a graphics window, as depicted in Figure A-1.

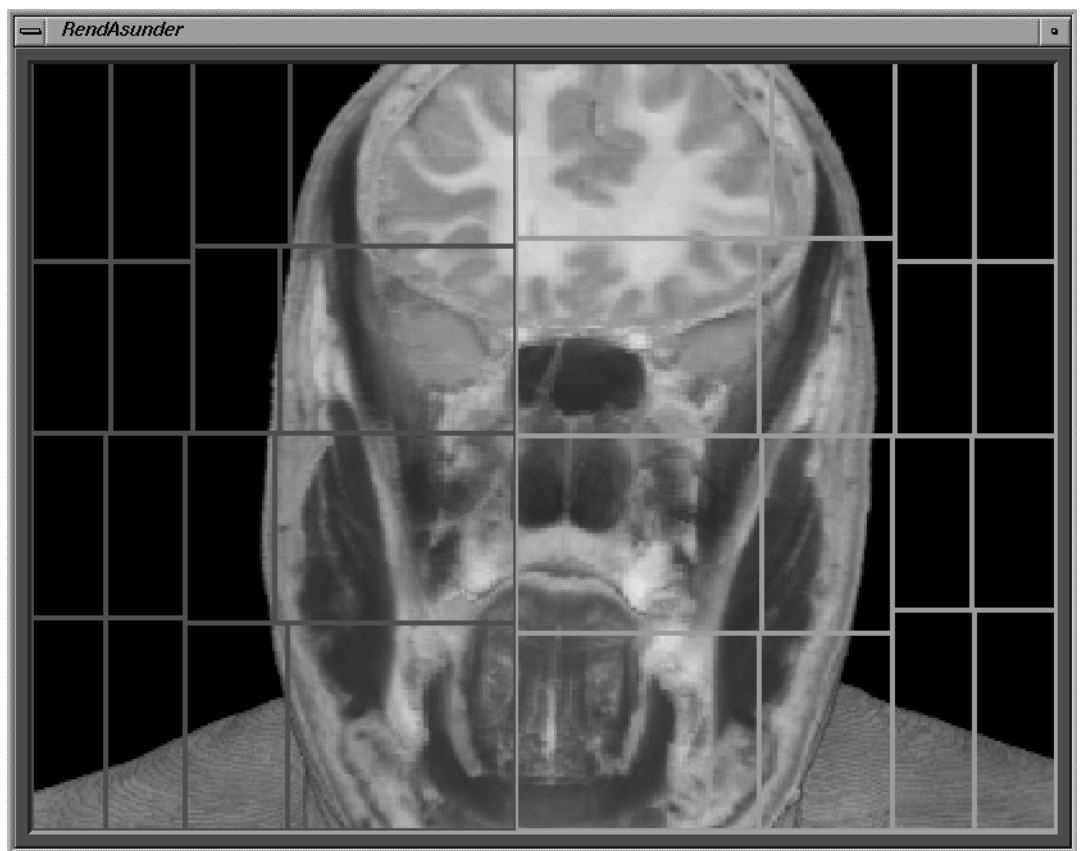


Figure A-1 RendAsunder Graphics Windows

The Graphics Window contains a rendering of the data from the current viewpoint, overlaid by rectangles that delineate the areas of the screen rendered by each processor. Different colored lines delineate areas that are rendered by different nodes. During execution, *RendAsunder* dynamically adjusts the regions of the image rendered by each node and processor to balance the workload.

The Controls Window

Figure A-2 shows the Controls menu.

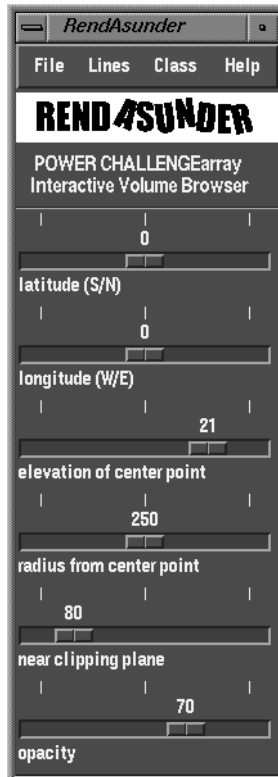


Figure A-2 RendAsunder Controls Menu

Controls Window Menus

The Controls Window contains the following menus:

- File menu
 - Home View option, which takes you back to the default view, in case you get lost

- five Save and Restore options, which allow you to save and restore five viewpoints

The viewpoints are not saved between sessions. If you restore a particular viewpoint before you have saved to it, *RendAsunder* displays one of the five default viewpoints.

You may find it interesting to look at these five default positions before you save over them. If you have saved over them and want to reset them to the defaults, use the Reset Default Stored Positions option.

- Record Movie and Play Movie options, which allow you to record a sequence of viewpoints, and play them back.
- Quit option, the best way to exit cleanly
- Class menu, which lets you select classes of data to be displayed by changing the color map in use. Select All Classes to render all data visibly. The other Class settings omit one or more types of data to produce different displays.
- Help menu, which displays information about the program (later versions will contain more extensive help)

Controls Window Sliders

The Controls Window contains a number of sliders that let you navigate around the data. Click on a slider to highlight it, and then use the left and right arrow keys to move the slider.

The sliders provide a convenient way to get smooth animated motion. Select one slider, scroll it with repeated key taps to the position you want, and then select another slider. For big jumps, drag the slider with the mouse.

Refresh the display by clicking on the currently highlighted slider.

The demo uses a geographical metaphor to navigate around the data. From any position you choose, you are always looking towards a center focus point in the middle of the data.

Latitude and Longitude

These two sliders allow you to move across lines of latitude in the north and south directions until you reach the poles, or move east and west across lines of longitude.

These controls allow you to reach any point on the surface of a sphere; recall that from the surface of this sphere you are always looking in the direction of the center focus point.

Elevation of Center Point

The center focus point is initially in the middle of the body. Use the elevation of center point control to raise and lower the center focus point.

For example, if you raise it to about 2.8, you will be looking at the head from the surface of a sphere centered about the head. If you lower it to about -3.0, you are centered about the feet. Any movement about the sphere orbits around this center point.

Radius From Center Point and Near Clipping Plane

Use the radius from center point control to zoom in and out from the center focus point. This control is often used in conjunction with the Near Clipping Plane control, which determines the distance of the near clipping plane from your eye.

If the value of the near clipping plane is much larger than the radius from center point, you might not be able to see anything because the entire volume will be clipped.

Opacity

Use this control to vary the opacity of the entire dataset. For low-opacity values, this control yields an X-ray effect.

For More Information

To learn more about RendAsunder, see the sources listed in X.

Table A-1 Information Sources

Topic	Book, Reference Page, or URL	Book Number
RendAsunder	http://www.scp.caltech.edu:80/~mep/work.html	
RendAsunder	http://www.nlm.nih.gov/research/visible/visible_human.html	

Array Documentation Quick Reference

This appendix lists all the information sources cited in the preceding chapters for convenient reference. The rows of the table are sorted alphabetically by the first column.

Table B-1 Information Sources

Topic	Book or URL	Book Number
Ada95 (GNU Ada Translator, GNAT)	<i>GNAT User's Guide</i>	007-2624-002
<i>ainfo</i> command	<i>ainfo</i> (1) and "Interrogating the Array" on page 37	
Applied Parallel Research	http://www.infomall.org/apri	
<i>array</i> command	use: <i>array</i> (1); configuration: <i>arrayd.conf</i> (4); "Using Array Services Commands" on page 33	
Array Services	Chapter 2, "Using an Array" <i>array_services</i> (5)	
Array Services simple configurator	<i>arrayconfig</i> (1m)	
Array Services daemon overview	<i>arrayd</i> (1m)	
Array Services Overview	<i>array_services</i> (5)	
<i>arshell</i> command	<i>arshell</i> (1) and "Using <i>arshell</i> " on page 42	
Assembly Language	<i>MIPSPro Assembly Language Programmer's Guide</i>	007-2418-002
Automatic parallelization of C and Fortran code	<i>MIPSpro Power Fortran 77 Programmer's Guide</i> <i>MIPSpro Power Fortran 90 Programmer's Guide</i> <i>IRIS Power C User's Guide</i>	007-2363-001 007-2760-001 007-0702-030
<i>aview</i> command	<i>aview</i> (1) and "Browsing With ArrayView" on page 40	

Table B-1 (continued) Information Sources

Topic	Book or URL	Book Number
C language	<i>C Language Reference Manual</i>	007-0701-090
C++ language	<i>C++ Programmers Guide</i>	007-0704-090
Center for Research in Parallel Computation	http://www.crpc.rice.edu/	
CHALLENGE and CHALLENGE 10000	<i>POWER CHALLENGE XL Rackmount Owner's Guide</i>	007-1735-040
CHALLENGEcomplib overview	http://www.sgi.com/Products/Challengecomplib.html	
IRIX Checkpoint and Restart (CPR)	<i>IRIX Checkpoint and Restart Operation Guide</i>	007-3236-xxx
Codine	http://www.instrumental.com	
Configuration file format	<code>arrayd.conf(4)</code> , <code>/usr/lib/array/arrayd.conf.template</code>	
<i>dbx, prof, pixie</i>	<i>dbx User's Guide</i>	007-0906-100
	<i>MIPS Compiling and Performance Tuning Guide</i>	007-2479-001
Debugger	<i>Developer Magic: Debugger User's Guide</i>	007-2579-002
Developer Magic	<i>Developer Magic: ProDev WorkShop Overview</i>	007-2582-003
	http://www.sgi.com/Products/WorkShop.html	
Developer Magic overviews	<i>Developer Magic: ProDev WorkShop Overview</i>	007-2582-003
	http://www.sgi.com/Products/WorkShop.html	
Directory of WWW software sources	http://www.yahoo.com/Computers_and_Internet/Software/Mathematics/	
Extreme Visualization Console	<i>POWER CHALLENGE XL Rackmount Owner's Guide</i>	007-1735-040
High Performance Fortran forum	http://www.crpc.rice.edu/HPFF/home.html	
High Performance Fortran texbook	<i>The High Performance Fortran Handbook</i> , Koelbel, Loveman, Schreiber, Steele Jr., and Zosel; MIT Press, 1994 (http://www-mitpress.mit.edu/)	ISBN 0-262-61094-9

Table B-1 (continued) Information Sources

Topic	Book or URL	Book Number
HIPPI Crossbar Switch	<i>EPS-1 User's Guide</i> http://www.esscom.com	09-9010
HIPPI interface	<i>IRIS HIPPI Administrator's Guide</i> <i>IRIS HIPPI API Programmer's Guide</i>	007-2229-003 007-2227-002
HPPC software collection	http://www.netlib.org/nhse/home.html	
HPCC Vendors "Mall"	http://www.npac.syr.edu/infomall/	
Index to math and statistical software	http://gams.nist.gov/	
IRISconsole	<i>IRISconsole Administrator's Guide</i> http://www.sgi.com/Products/hardware/challenge/IRISconsole.html	007-2872-001
IRISconsole hardware	<i>IRISconsole Installation Guide</i> <i>Indy Workstation Owner's Guide</i>	007-2839-001 007-9804-040
IRIX 6.2 Data Sheet	http://www.sgi.com/Products/software/IRIX6.2/IRIX62DS.html	
IRIX 6.2 Specifications	http://www.sgi.com/Products/software/IRIX6.2/IRIX62specs.html	
IRIX IPC facilities	<i>Topics In IRIX Programming</i>	007-2478-003
HIPPI Interconnect	<i>IRIS HIPPI API Programmer's Guide</i>	007-2229-003
Listing and Monitoring Processes	ps(1), top(1), and gr_top(1); gr_osview(1), gmemusage(1)	
Load Sharing Facility	http://www.platform.com	
Logging in to a node	Chapter 2, "Using an Array"	
Memory hierarchies; locality of reference	<i>Computer Architecture: A Quantitative Approach</i>	ISBN 1-55860-069-8
MIPSpro compiler features and use	<i>MIPS Compiling and Performance Tuning Guide</i>	007-2479-001

Table B-1 (continued) Information Sources

Topic	Book or URL	Book Number
MIPSpro Fortran 77	<i>MIPSpro Fortran 77 Programmer's Guide</i>	007-2361-002
	<i>MIPSpro Fortran 77 Language Reference Manual</i>	007-2362-002
MIPSpro Fortran 90	<i>MIPSpro Fortran 90 Programmer's Guide</i>	007-2761-001
Models of Parallel Computation	<i>Topics In IRIX Programming</i>	007-2478-003
XMPI and XPVM	<i>MPI and PVM User's Guide</i> mpirun(1)	007-3286-xxx
Message Passing Toolkit (MPT) in general	http://www.cray.com/PUBLIC/product-info/sw/	
MPI Overview	mpi(5)	
MPI References	<i>Using MPI</i> , Gropp, Lusk, and Skjellum, MIT Press 1995 (http://www.mitpress.mit.edu/) <i>Using MPI</i> (in IRIX Insight library)	ISBN 0-262-69184-1007-2855-001
	<i>MPI, The Complete Reference</i> , Snir, Otto, Huss-Lederman, Walker, and Dongarra, MIT Press 1995	ISBN 0-262-57104-8
MPI Standard	http://www.mcs.anl.gov/mpi	
MPI and PVM jobs	<i>MPI and PVM User's Guide</i>	007-3286-001
newsess command	newsess(1)	
NetVisualyzer	NetVisualyzer User's Guide	007-0812-040
Network Queuing Environment (NQE) technical papers	http://www.sdiv.cray.com/~nqe/nqe_external/index.html (pointers to technical papers)	
	http://www.cray.com/PUBLIC/product-info/sw/nqe/nqe30.html (illustrated overview)	
	<i>NQE User's Guide</i>	SG-2148 3.2
	<i>NQE Administrator's Guide</i>	SG-2150 3.2
Numerical software	http://www.netlib.org/	
Origin2000 and Origin200	http://www.sgi.com/Products/hardware/servers/index.html	

Table B-1 (continued) Information Sources

Topic	Book or URL	Book Number
Onyx2 and RealityMonster	http://www.sgi.com/Products/hardware/graphics/products/index.html	
Parallel Programming Models Compared	<i>Topics In IRIX Programming</i>	007-2478-003
Pascal	Pascal Programming Guide	007-0740-030
PerfAcct	http://www.instrumental.com	
Performance Analysis	<i>Developer Magic: Performance Analyzer User's Guide</i>	007-2581-002
Performance Analysis	<i>Developer Magic: Static Analyzer User's Guide</i>	007-2580-002
Performance Analysis (command-line tools)	<i>MIPSpro Compiling and Performance Tuning Guide</i>	007-2360-003
Performance Co-Pilot	<i>The Performance Co-Pilot User's and Administrator's Guide</i>	007-2614-001
	<i>Performance Co-Pilot for Informix-7 User's Guide</i>	007-3007-001
Performance Co-Pilot data sheet	http://www.sgi.com/Products/hardware/challenge/CoPilot/CoPilot.html	
Performance Tuning on Origin2000 and Onyx2	<i>Performance Tuning Optimization for Origin2000 and Onyx2</i> , online only at http://www.sgi.com/techpubs/lib/makepage.cgi?007-3430-001	007-3430-xxx
Portland Group, Inc.	http://www.pgroup.com	
POWER CHALLENGE	<i>POWER CHALLENGE XL Rackmount Owner's Guide</i>	007-1735-040
POWER Onyx	<i>POWER Onyx and Onyx Rackmount Owner's Guide</i>	007-1736-060
Process ID and process group	intro(2) — scan to the section headed "Definitions"	
<i>prof, pixie</i>	<i>dbx User's Guide</i>	007-0906-100
	<i>MIPS Compiling and Performance Tuning Guide</i>	007-2479-001
PVM Home Page	http://www.epm.ornl.gov/pvm/pvm_home.html	
PVM Overview	pvm(1PVM)	
PVM to MPI	<i>Topics In IRIX Programming</i>	007-2478-003

Table B-1 (continued) Information Sources

Topic	Book or URL	Book Number
PVM Reference	<i>PVM: Parallel Virtual Machine</i> , Geist, Beguelin, Dongarra, Weicheng Jiang, Manchek, and Sunderam, MIT Press 1994 http://www.netlib.org/pvm3/book/pvm-book.html	ISBN 0-262-57108-0
PVM and MPI jobs	<i>MPI and PVM User's Guide</i>	007-3286-001
REACT/pro and real-time programming	http://www.sgi.com/real-time/	
RealityEngine ² and InfiniteReality	http://www.sgi.com/Products/hardware/Onyx/Tech/	
Remote login	rlogin(1)	
RendAsunder	http://www.scp.caltech.edu:80/~mep/work.html	
RendAsunder	http://www.nlm.nih.gov/research/visible/visible_human.html	
Running programs at low priority	nice(1), batch(1)	
Running programs at a scheduled time	at(1)	
Setting environment variables	environ(5), env(1)	
SGI Servers	http://www.sgi.com/Products/index.html?hardware	
Shared-memory communication	<i>Topics in IRIX Programming</i> (chapter 1 and 2)	007-2478-003
Software pipelining	MIPSpro 64-Bit Porting and Transition Guide	007-2391-002
Software pipelining	MIPSpro Compiling and Performance Tuning Guide	007-2360-003
Terminating a process	kill(1)	
Visualization tools for physics	http://www.lassp.cornell.edu/LASSPTools/LASSPTools.html	
Volume Renderer and other tools	http://www.arc.umn.edu/gvl-software/gvl-software.html	

Tell Us About This Manual

As a user of Silicon Graphics products, you can help us to better understand your needs and to improve the quality of our documentation.

Any information that you provide will be useful. Here is a list of suggested topics:

- General impression of the document
- Omission of material that you expected to find
- Technical errors
- Relevance of the material to the job you had to do
- Quality of the printing and binding

Please send the title and part number of the document with your comments. The part number for this document is 007-3058-003.

Thank you!

Three Ways to Reach Us

- To send your comments by **electronic mail**, use either of these addresses:
 - On the Internet: techpubs@sgi.com
 - For UUCP mail (through any backbone site): *[your_site]!sgi!techpubs*
- To **fax** your comments (or annotated copies of manual pages), use this fax number: 415-965-0964
- To send your comments by **traditional mail**, use this address:

Technical Publications
Silicon Graphics, Inc.
2011 North Shoreline Boulevard, M/S 535
Mountain View, California 94043-1389