P-75

NASA Technical Memorandum 107599 (Revised)

CRAY

Mini Manual
January 1993

Central Scientific
Computing Complex

Document CR-1d

NASA
National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-0001

NASA Langley Research Center

Central Scientific
Computing Complex
Document CR-1d

CRAY Mini Manual

January 1993

(Replaces CR-1c Dated April 1992)

Geoffrey M. Tennille and Lona M. Howser

PREFACE

This record of revision is a permanent part of the *CRAY Mini Manual.* In this release, some section headings have the suffix [d] added to indicate that the section was added or substantially changed at this revision. Section headings without a suffix or with suffixes [a] through [c] have not changed since the last revision. This method of annotation was chosen since sections in the *CRAY Mini Manual* are generally brief, thus a revision record can be kept with the text of the document. Grammatical, spelling, section renumbering and other changes that don't effect the context of the section are not marked.

DATE	REVISION	MAJOR FEATURES
March 1989	Preliminary	
July 1989	Original	Corrections to Preliminary version; NQS queues; MSS Utilities
March 1990	Revision a	UNICOS 5.1; File System Quotas; Autotasking; Appendix on Unsupported Software; cdbx debugger; New options for MSS utilities; New syntax for cf77 command; List of acronyms used; Section guides to hidden files and commands used in this manual
February 1991	Revision b	Reorganization of manual; Installation of CRAY Y-MP; FORGE; hpm; perftrace; Standard C; Additional information on autotasking, cdbx and quotas; Automatic symbolic dump (debugx)
April 1992	Revision c	Hardware upgrades to Voyager and Sabre; UNICOS 6.1; cf77 Version 5; Standard C Version 3; Removal of dda and drd debuggers; New X-based tools for autotasking; Section on run-time errors
January 1993	Revision d	Hardware upgrade to Sabre; Solid-state Storage Device; New NQS queues; Removal of FORGE from Voyager; Mixing C and FORTRAN.

ACRONYMS [d]

ACD	Analysis and Computation Division
ANSI	American National Standards Institute
ARC	Ames Research Center
BLAS	Basic Linear Algebra Subroutines
CAB	Computer Applications Branch
CAL	CRAY Assembly Language
CGL	Common Graphics Library
CMB	Computer Management Branch
CPU	Central Processing Unit
DCM	Division Computing Manager
DRAM	Dynamic Random-Access Memory
EARS	Explicit Archival and Retrieval System
FTP	File Transfer Protocol
GAS	Graphics Animation System
IBM	International Business Machines
IMSL	International Mathematical Statistical Library
I/O	Input/Output
ISO	International Organization for Standardization
LaRC	Langley Research Center
LaTS	LaRC Telecommunications System
LCUC	LaRC Computer Users Committee
MFLOPS	Million FLoating Operations Per Second
MOTD	Message-of-the-Day
MSS	Mass Storage Subsystem
NAS	Numerical Aerodynamic Simulator
NCAR	National Center for Atmospheric Research
NCS	NOS Computing Subsystem
NFS	Network File System
NOS	Network Operating System
NQS	Network Queuing System
NSO	Network Support Office
OCO	Operations Control Office
PVI	Precision Visuals, Inc.
RGL	Remote Graphics Library
RM/RMT	Raster Metafile/RM Translator
SAG	Supercomputing Applications Group
SCCS	Source Code Control System
SNS	Supercomputing Network Subsystem
SRAM	Static Random-Access Memory
SSD	Solid-state Storage Device
SSIG	Supercomputing Special Interest Group
TCP/IP	Transmission Control Protocol/Internet Protocol
X	X Window System

Table of Contents

1. INTRODUCTION [b]

Langley Research Center's (LaRC's) supercomputers, the CRAY-2S, **Voyager,** and the CRAY Y-MP, **Sabre,** provide a significant portion of the Center's computing resources. If you are an experienced UNIX user, it is recommended that you peruse this manual to check for topics unique to CRAY computers prior to using either **Voyager** or **Sabre.** If you are either a novice UNIX user or unfamiliar with the LaRC environment, you should also read Document A-8, the *SNS Programming Environment User's Guide* prior to using **Voyager** or **Sabre.**

1.1. The CRAY Mini Manual [d]

The purpose of the *CRAY Mini Manual* is to provide **Supercomputing Network Subsystem (SNS)** users with basic information about **Voyager** and **Sabre,** the two CRAY supercomputers installed at NASA Langley Research Center (LaRC).

This revision reflects the hardware upgrades to LaRC's CRAY Y-MP, **Sabre,** (See section 1.3) and installation of a Solid-state Storage Device (see Section 4.4). Additionally, FORGE (See section 6.5) has been removed from **Voyager.** FORGE, however is still available for Sun Workstations. The structure of NQS queues (See section 4.3) has changed. Information on mixing C and FORTRAN code has been added (See section 3.2). Within the Mini Manual, new sections and sections that have changed substantially are marked with the revision level, [d], in the section title. References to other manuals omit revision levels. See **Table 7.1** for the revision level of CRAY documentation associated with UNICOS 6.1.6. Questions, comments and suggestions about this manual are solicited from the user community.

This mini manual does not attempt to provide a tutorial on any given topic. It is intended as a broad overview of the system, with references to more detailed information. Commands described in this manual usually have several other options for which space does not allow a discussion. Examples are for the C-shell unless specifically designated as Bourne shell examples. Use of the *man* pages, a standard UNIX feature, is recommended to learn more about any particular command. The *notes* utility on the CONVEX computers, **Eagle** and **Mustang,** is used to disseminate information rapidly to SNS users. Another useful resource is the *CRAY UNICOS Primer,* Document CR-2 (See section 7.1). It provides more detailed examples than the *CRAY Mini Manual.*

1.2. Characteristics of Voyager [b]

The official designation of LaRC's CRAY-2, **Voyager,** is a CRAY-2S/4-128. Hereafter it will be designated simply as CRAY-2S. The 'S' indicates that the memory is static rather than dynamic and the '4-128' indicates that **Voyager** has four processors and 128 million 64-bit words of central memory. The following sections give a brief description of the major hardware and performance features of the CRAY-2S.

1.2.1. CRAY-2S Hardware [c]

The CRAY-2S is a register to register vector processor with 4 identical and independent background processors and a foreground processor. It has 128 Mwds of Static Random Access Memory (SRAM), interleaved in 128 memory banks. Each background processor also has a 16,384 word local memory. The CRAY-2S has no virtual memory. The scalar and vector capability of this machine in a multiprocessing environment produce extremely high result rates. The entire mainframe, which includes all memory, computer logic and DC power supplies, is integrated into a compact package consisting of 14 vertical columns, each 4 foot high, arranged in a 300 degree arc. Cooling is provided by fluorinert, an inert noncorrosive liquid, which circulates within the mainframe in direct contact with the integrated circuit packages. Additionally, **Voyager** is now configured with 37 Gbytes of disk storage on 2 DS-41 disk drives.

Each of the four independent background processors has eight 64 word vector registers, eight scalar and eight address registers. Each processor has a single port to the large common memory, through which vectors are transferred to the vector registers. There are four vector functional units, three scalar functional units and two address functional units. It is possible for these functional units to be operating concurrently on independent operands. Chaining is not supported on **Voyager** like it is on the CRAY X-MP and CRAY Y-MP. A set of eight semaphore flags allows for communication and synchronization between the background processors. One flag is assigned to each background CPU and one is assigned to each currently active process.

The local memory is used to hold scalar operands during a period of computation. It can also be used for the temporary storage of vector operands, when they are used more than once during a computation in the vector registers.

The foreground processor controls and monitors system operations and includes high-speed synchronous communications channels which interconnect the background CPU's, the foreground processor, disk, HSX and external I/O controllers. The HSX channel controller connects high-speed external devices to the CRAY-2S system. The foreground processor also responds to background CPU requests and sequences channel communications signals.

1.2.2. CRAY-2S Performance [a]

The CRAY-2S, **Voyager,** is a 4 CPU vector and parallel computer. Each CPU is a powerful register-to-register vector processor (4.1 nanosecond minor cycle) capable of generating results in the 100-300 MFLOPs (Million FLoating OPerations per Second) range. Additionally, the four processors can be brought to bear on a single problem, called multitasking, to greatly increase that performance. On one problem a rate of over 1 GigaFLOP (one billion floating point operations per second) has been achieved. The vector speed of the computer is accomplished primarily through the vectorization capability of the compiler. Generally, this is the optimization of an innermost DO loop. However, the programmer can significantly affect performance through program design and programming techniques employed.

The MFLOP speed of **Voyager** is primarily dependent upon three things:

1. Vector length - Vector length is defined to be the length of the DO loop that has been vectorized by the compiler. In actuality, for loops of length greater than 64 the compiler generates code to process the loop in groups of 64 since that is the size of each vector register and hence the maximum number of operands that can be involved in any one vector instruction. Even though a vector register can only process 64 elements at a time, performance does increase with increasing vector length to a maximum (called r-infinity) for "infinite " length vectors. Experience to date indicates that vectors in the 20 - 40 length range achieve half the performance of r-infinity and vectors of length 64 achieve 60-75 percent of r-infinity if the loop is computationally intensive (See dependency 3.). The actual figures vary with the type of calculation being performed.

2. Vector stride - Vector stride is defined to be the separation in memory between elements of a vector. For instance, the vector A(2), A(6),A(10), etc. has a stride of 4 . Vectors with unit or other odd stride can be loaded from memory to a vector register at a rate of one element per minor cycle after an initial startup of approximately 35 cycles. However, due to the division of the memory into four quadrants and each quadrant into 32 banks, the transfer of vectors with any even stride suffer some degradation in performance. The worst strides, in order of increasing degradation are multiples of 2, 4, 64, and 128. The cycles per element for these strides are 2,4,6, and 12 respectively.

Table 1.1 gives MFLOP rates for a vector addition of length 64 as a function of of the stride. Since the largest percentage of time in this loop is the data transfer to and from memory, the rates are greatly affected by the stride. In a kernel with more calculations per memory reference, the load/store time has a less dramatic effect on the overall performance.

	Stride									
Vector Length	1	2	3	4	8	16	32	64	128	256
N=64	23.0	12.1	19.6	8.6	8.6	8.6	8.6	7.5	4.3	4.3

Table 1.1 - Effect of Stride on CRAY-2S MFLOP Rate of Vector Addition

3. Computational richness of the loop - The CRAY-2S CPU is extremely fast in carrying out its floating point operations once operands are in the vector registers. However, if much of the time spent in executing a DO loop is devoted to loading (or storing) of vector operands from memory, that performance is degraded. Consequently, a very simple loop, such as:

C(I) = A(I) + B(I) , I=1,N

where the ratio of memory activity to computation is 3 to 1, has an r-infinity (See discussion of vector length on previous page) of only 67 MFLOPS, and executes at 23 MFLOPS for vectors of length 64. On the other hand, a loop such as:

A(I) = B(I)*C(I) + D(I)*E(I) + 3.*(F(I) - G(I))*(F(I) +H(I)) , I=1,N

which has a 1 to 1 ratio, has an r-infinity of 137 MFLOPs, and achieves 104 MFLOPS for N=64.

The richness of the latter loop also allows increased performance in other ways. The CRAY-2S allows some functional units to execute in parallel. For instance, where there is no conflict in the use of the vector registers, it is possible for a vector load (or store), vector multiplication, and vector addition to be in execution simultaneously. The compiler attempts to schedule its instructions in such a way to maximize this type of activity. However, in loops in which very little computation occurs, there is little or no opportunity for overlap.

Benchmark results on a collection of application programs show **Voyager** executing 5.9 times as fast as **Eagle** or **Mustang** and approximately 25 percent faster than **Navier,** the 256 Mwd CRAY-2D at NASA Ames Research Center (ARC). This speed difference is primarily due to improved performance of the SRAM memory on **Voyager,** as compared to the Dynamic Random Access Memory (DRAM) of **Navier.**

1.3 Characteristics of Sabre [d]

The official designation of LaRC's CRAY Y-MP **Sabre** is a CRAY Y-MP8E/8256. Hereafter it will be designated simply as CRAY Y-MP. The '8E' indicates that **Sabre** is a model 'E' machine with an eight CPU chassis. The '8256' indicates that eight processors are installed and configured with 256 million words (Mwds) of memory. The following sections give a brief description of the major hardware and performance features of the CRAY Y-MP.

1.3.1 CRAY Y-MP Hardware [d]

The CRAY Y-MP is a register to register vector processor. The configuration of **Sabre** is 8 identical and independent processors with interprocessor communication and an I/O Subsystem (IOS). There is a 512 Mwd Solid-state Storage Device (SSD) (See section 4.4) associated with **Sabre.** It has 256 Mwd of SRAM central memory that is shared by the processors and the IOS. Memory is divided into 256 interleaved banks, which improves the speed of memory access by allowing simultaneous and overlapping memory references. Each CPU has four parallel ports to central memory, each of which performs specific functions. Bidirectional access to memory allows both block reads and writes to be done simultaneously. The CRAY Y-MP has no virtual memory and unlike **Voyager,** does not have local memory associated with each processor. The scalar and vector capability of this machine in a multiprocessing environment produce extremely high result rates.

The entire mainframe, which includes all memory, computer logic and DC power supplies, is integrated into a compact package that is about six and one half feet tall and covering an area of about sixteen square feet. Cooling is provided by fluorinert, an inert noncorrosive liquid, which circulates through each module, power supply and power supply mounting plate but not in direct contact with the integrated circuit packages like **Voyager. Sabre** is now configured with 80 Gbytes of disk storage on 2 DD-4R disk drives.

Each of the independent processors has eight 64 word vector registers, eight scalar and eight address registers. Additionally, the scalar and address registers each have sixty-four intermediate registers. Each processor has four parallel ports to the common memory, through which vectors are transferred to the vector registers. There are five vector functional units, four scalar functional units and two address functional units. It is possible for these functional units to be operating concurrently on independent operands. Chaining is supported on **Sabre.** The CRAY Y-MP also has a built-in performance monitor, *hpm* (See section 6.1.2), that allows a programmer to gather performance data in an efficient manner.

The I/O Subsystem (IOS) has several I/O Processors (IOP's), a buffer memory and necessary interfaces. It is designed for fast data transfer between buffer memory and central memory, as well as to peripheral devices. The IOS also interfaces with the

High-speed External Communications (HSX) channel. The HSX channel controller connects high-speed external devices to the CRAY Y-MP system.

1.3.2 CRAY Y-MP Performance [d]

LaRC's CRAY Y-MP, **Sabre,** is an 8 CPU vector and parallel computer. Each CPU is a powerful register-to-register vector processor (6 nanosecond minor cycle) capable of generating results in the 150-300 MFLOPS (Million FLoating OPerations per Second) range. Additionally, the eight processors can be brought to bear on a single problem, called multitasking, to greatly increase that performance. The vector speed of the computer is accomplished primarily through the vectorization capability of the compiler. Generally, this is the optimization of an innermost DO loop. However, the programmer can significantly affect performance through program design and programming techniques employed.

The MFLOP speed of **Sabre** is primarily dependent upon four things:

1. Vector length - Vector length is defined to be the length of the DO loop that has been vectorized by the compiler. In actuality, for loops of length greater than 64 the compiler generates code to process the loop in groups of 64 since that is the size of each vector register and hence the maximum number of operands that can be involved in any one vector instruction. Even though a vector register can only process 64 elements at a time, performance does increase with increasing vector length to a maximum (called r-infinity) for "infinite " length vectors. Experience to date indicates that vectors in the 75 - 100 length range achieve half the performance of r-infinity and vectors of length 64 achieve 40-45 percent of r-infinity if the loop is computationally intensive (See dependency 3.). The actual figures vary with the type of calculation being performed.

2. Vector stride - Vector stride is defined to be the separation in memory between elements of a vector. For instance, the vector A(2), A(6),A(10), etc. has a stride of 4 . Vectors with unit or other odd stride can be loaded from memory to a vector register at a rate of one element per minor cycle after an initial startup of approximately 15 cycles. However, due to the division of the memory into four sections of 64 banks each, the transfer of vectors with any non-unit stride suffers some degradation in performance. The worst strides, in order of increasing degradation, are multiples of 8, 16 and 32. Strides of a multiple of 2 or 4 are no more of a detriment to performance than odd strides. The average cycles per element for the worst strides are 1.25, 2.5 and 5 respectively.

Table 1.3 gives MFLOP rates for a vector addition of length 64 as a function of of the stride. Since the largest percentage of time in this loop is the data transfer to and from memory, the rates are greatly affected by the stride. In a kernel with more calculations per memory reference, the load/store time has a less dramatic effect on the overall performance.

	Stride									
Vector Length	1	2	3	4	8	16	32	64	128	256
N=64	21.0	18.4	18.4	18.4	16.2	13.0	9.3	9.3	9.3	9.3

Table 1.3 - Effect of Stride on CRAY Y-MP MFLOP Rate of Vector Addition

3. Computational richness of the loop - The CRAY Y-MP CPU is fast in carrying out its floating point operations once operands are in the vector registers. However, if much of the time spent in executing a DO loop is devoted to loading (or storing) of vector operands from memory, that performance is degraded. Consequently, a very simple loop, such as:

C(I) = A(I) + B(I) , I=1,N

where the ratio of memory activity to computation is 3 to 1, has an r-infinity (See discussion of vector length on previous page) of only 138 MFLOPS, and executes at 21 MFLOPS for vectors of length 64. On the other hand, a loop such as:

A(I) = B(I)*C(I) + D(I)*E(I) + 3.*(F(I) - G(I))*(F(I) +H(I)) , I=1,N

which has a 1 to 1 ratio, has an r-infinity of 230 MFLOPs, and achieves 104 MFLOPS for N=64.

The richness of the latter loop also allows increased performance in other ways. The CRAY Y-MP allows some functional units to execute in parallel. For instance, where there is no conflict in the use of the vector registers, it is possible for a vector load (or store), vector multiplication, and vector addition to be in execution simultaneously. The compiler attempts to schedule its instructions in such a way to maximize this type of activity. However, in loops in which very little computation occurs, there is little or no opportunity for overlap.

4. Chaining - The multiple ports to memory of each of the CRAY Y-MP's processors allows for the overlapping of memory accesses. The add and multiply functional units may also be kept busy at the same time. Consider the following simple loop:

$$C(I) = S * (A(I) + B(I)), I= 1,N$$

Because of the multiple ports to memory, after the access of the array A is initiated, the access to the array B can begin after just one clock cycle. Then once an element of both A and B have been loaded into the vector registers, the addition can begin. After the addition of the first elements of A and B is complete, that result can be chained to the multiply functional unit to be multiplied by the scalar S. When the first result of that operation is complete, a third path to memory may be used to begin the store of the array C. On **Sabre,** the computational time for this loop is O(N), since the three memory accesses (two loads and a store) can be overlapped with the two arithmetic operations. On **Voyager,** the computational time for the loop is O(5N), since none of the memory accesses or arithmetic operations can be overlapped.

1.4. CRAY Software [c]

The CRAY UNICOS operating system is derived from the AT&T System V UNIX and has been enhanced for use on a supercomputer. The version currently installed is UNICOS 6.1. It supports most traditional features of UNIX, some of which are described in sections 2.2.5, 2.2.7 and 3 of the *SNS Programming Environment User's Guide* (Document A-8). The NQS batch facility (See section 4.3.1) provides true batch functionality in a system designed for interactive access. There are two FORTRAN compilers (See chapter 2) as well as PASCAL and two C compilers (See sections 3.1, 3.2, and 3.3). The FORTRAN compilers automatically vectorize user source codes. There is no explicit vector syntax.

Several mathematical libraries (See section 7 of A-8), debugging utilities (See section 5), graphics packages (See section 6 of A-8) and code management utilities (See section 6) are supported on **Voyager** and **Sabre.** Additionally UNICOS supports the Transmission Control Protocol/Internet Protocol (TCP/IP) utilities (See section 5 of A-8). Multitasking from within FORTRAN programs is supported both automatically and via compiler directives (See section 6.3).

1.5. CRAY Programming [b]

CRAY supports three higher level programming languages: FORTRAN, PASCAL and C. There are two different FORTRAN compilers: *cft77* and *cft,* both of which are based on the ANSI FORTRAN X3.9-1978 standard (FORTRAN 77). Use of the *cft77* compiler is recommended, since it is supported across the entire CRAY product line and generally produces more efficient executable code. The *cft* FORTRAN compiler is not available on **Sabre.** See section 2.1 for more detailed information on the FORTRAN compilers.

The PASCAL compiler is called *pascal* (See section 3.1). CRAY PASCAL is a complete implementation of the Level 1 requirements of PASCAL standard ISO 7185 (as defined by the International Organization for Standardization). Document CR-21 is the CRAY PASCAL Reference Manual. The book **PASCAL User Manual and Report (Second Edition),** by Kathleen Jensen and Nicklaus Wirth is considered a standard PASCAL reference manual.

The C compilers are called *cc* and *pcc* (See sections 3.2 and 3.3). The *cc* compiler is the Standard C compiler and the *pcc* compiler is the portable C compiler, which is based on the AT&T implementation. Documents CR-30 and CR-20 are the *CRAY* Standard and *CRAY C Reference Manual* respectively. **The C Programming Language,** by Brian Kernighan and Dennis Ritchie is considered the standard C programming language reference manual by most C programmers. Version 2 of Kernighan and Ritchie describes Standard C.

2. FORTRAN ON THE CRAYS [c]

Both CRAY FORTRAN compilers are an implementation of FORTRAN 77, as defined by the American National Standards Institute (ANSI 3.9-1978) with extensions. The *cft77* compiler contains the full FORTRAN 77 standards, offers automatic vectorizing of code and other automated features designed to exploit the CRAY hardware. On **Voyager,** the older compiler *cft* is supported at a maintenance level only, and will not be supported after UNICOS 6. It is also not available on **Sabre** and is no longer described in this manual.

The *cft77* compiler is not very fast. It is recommended to use the *make* utility (See section 3.3 of A-8) to only recompile those routines that have been modified, rather than the entire code. Using *make* to recompile only those routines that have been modified, is a more efficient use of computer resources. Object code generated by the two compilers is however load compatible, so subroutines compiled with *cft* and *cft77* can be combined at load to create an executable.

2.1. Compiling with the cft77 Compiler [c]

The basic syntax for the *cft77* compiler is:

cft77 [-options] filename.f

The name of the file which contains your source code is *filename.f,* where the *.f* extension is required for the compiler to recognize the file as FORTRAN source code. **Executable files generated using *cft77* on Voyager and Sabre are not compatible.** You must recompile any source code that is moved between the two CRAY's.

Keywords may be in any order, separated by spaces. Spaces between a keyword and the argument are optional. The keywords *-e* (enable) and *-d* (disable) can be followed by a string concatenating the options. For a detailed description of *options,* use *man cft77* or see Chapter 1 of Document CR-35, *CF77 Compiling System, Volume 1: FORTRAN Reference Manual.*

Some of the more useful options and defaults are listed below.

-a alloc	Specifies memory allocation method. *alloc* can be: *static* - All memory allocated statically. Address location does not change. **(Default)** *stack* - Variables in SAVE, DATA and COMMON statements are allocated statically, all others are stack. **(Required for multitasking.)**
-b binfile	Binary object code of the program ready for the loader is written to *binfile.* **(Default is filename.o)**
-I inlinef	Subprograms contained in the file *inlinef* are expanded inline.
-l listfile	Listings enabled by the compiler are written to *listfile.* **(Default is filename.l. All listings are off by default.)**
-o optim	Specifies optimization level. *optim* can be: *noaggress, bl, noinline, noloopalign, recurrence, norecursive, scalar, vector, vsearch, nozeroinc.* **Default is vector.** Multiple optimization levels are separated by commas. See man pages or chapter 1 of CR-35 for specific details.
-N col	Specifies number of characters of source code to read. *col* can be 72 **(default)** or 80.

-R	Enables run-time checking of array bounds.
-ec	Enables a list of common block names and lengths.
-ef	Generates flowtrace for the entire compilation unit.
-ei	Causes uninitialized memory to be set to an undefined value.
-ej	Causes at least one execution of each DO loop when the DO statement is executed.
-em	Enables the Loopmark option, which marks each DO loop in the source listing and indicates type: vector, short vector or scalar.
-es	Listing of source code is written to *listfile.* **(Default is filename.l)**
-ex	Enables cross reference listing to be written to *listfile.* **(Default is filename.l)**
-ez	Enables use of debugging by generating a debug table.

Examples:

cft77 prog.f

File *prog.f* is compiled using all defaults; no listing is generated; the binary to be loaded is written to *prog.o;* and all optimization and vectorization is done.

cft77 -efmx prog.f

File *prog.f* is compiled with all optimization and vectorization. Flowtrace is enabled, so timing information can be generated at execution. File *prog.l* contains the source listing with each DO loop marked by type and cross reference listing. The binary to be loaded is written on file *prog.o.*

2.1.1. cft77 Differences On Voyager and Sabre [c]

There are some subtle differences in the implementation of FORTRAN on **Voyager** and **Sabre.** These differences should not prevent most applications from executing correctly. Check under *notes CRnews* on **Eagle** for all differences between **Voyager** and **Sabre.** Also, since the two machines have differing instruction sets, executable files generated using *cft77* are not compatible between them.

2.2. Loading With segldr [c]

The *segldr* command links relocatable binaries together to produce an executable binary. It invokes the loader and loads all libraries needed for FORTRAN. Information is passed to *segldr* by the use of options or directives. The syntax is:

segldr [options] files

The files listed in *files* are of the form:

filename.o	Binary object file generated by a compiler.
filename.a	Binary library file.
any other name	The files contain directives to segldr.

For a detailed description of the options and directives for *segldr,* use *man segldr* or see Document CR-8, *CRAY Segment Loader (SEGLDR) and ld Reference Manual.*

A few useful options are listed below:

-o outfile	Writes executable program to *outfile.* **(Default is a.out)**
-m	Writes an address level load map to stdout.
-l lib	Non-default libraries in directory lib are loaded, such as: *larclib, imslib, perf.*

Examples:

segldr prog.o

Creates an executable *a.out* from the binary object file *prog.o.*

segldr -o prog prog.o

Creates an executable *prog* from the binary object *prog.o.*

segldr -o prog -lflow prog.o

Creates an executable *prog* which includes the flowtrace libraries (See section 6.1.1) from the binary object file *prog.o.*

2.3. Using cf77 to Compile and Load [c]

The command *cf77* can be used to invoke the *cft77* compiler and the loader *segldr* The syntax of the *cf77* command allows options to be passed to various system phases in addition to many other options. Some of the most useful are listed here:

cf77 [-Z phase] [-c] [-F] [-l lib] [-o outfile] [-Wf"cft77_options"] files

The *files* have names of the form:

file.a	library input file
file.f	FORTRAN source file
file.o	binary object file

cft77_options	*cft77_options* can be used in the same manner they are used on the *cft77* command line. Multiple options are separated by white space within the double quotes.
-c	Produces .o object files, but does not produce an executable.
-F	Enables flowtrace processing, turns on *cft77* flowtrace option and loads flowtrace library.
-l lib	Non-default libraries in directory lib are loaded, such as: *larclib, imslib, perf.*
-o outfile	Executable binary file is written to *outfile.* **(Default is a.out.)**
-Z phase	Specifies code generation option, usually needed only when multitasking; *phase* can be one of the following:

c	- Activates FORTRAN compiler and loader only. **(Default)**
m	- Specifies microtasking
p	- Specifies autotasking

Additional options may be passed to the individual system phases as:

-Wd"fpp"	Dependence analyzer
-Wu"fmp"	Translator
-Wa"as"	Assembler
-Wl"segldr"	Loader
-Wp"gpp"	Generic preprocessor

If any of the multitasking options are specified, *cf77* ensures that the *stack* allocation option is selected. See section 6.3 and CR-38 for details about multitasking.

Examples:

*cf77 -c *.f*

All FORTRAN source code files in your current working directory are compiled to create object code files, but no executable file is generated.

cf77 -o prog -F - Wf"-emx" prog.f

The file *prog.f* is compiled with the *cft77* compiler and a source listing *prog.l* is generated. File *prog.l* has DO loops marked according to type and a cross reference. The flowtrace option is turned on and the flowtrace library is loaded. The executable created is called *prog*.

cf77 -o prog -Z p prog.f

The file *prog.f* is autotasked (See section 6.3.3) and an executable file, *prog,* is created.

2.4. Execution of FORTRAN Programs [a]

To execute your program after it has been loaded by the compiler, type the name of the executable file. The default name of the executable file is *a.out.*

FORTRAN unit numbers (except unit 5 and 6) are usually associated with a file by a FORTRAN OPEN statement. If a FORTRAN unit number is not associated with a file name by a FORTRAN OPEN statement, the default file name is *fort.n* where *n* is the unit number in the FORTRAN I/O statement. FORTRAN units 5 and 6 are standard input and output respectively. See section 2.4 of A-8 for redirecting I/O to and from units 5 and 6.

A set of sample commands which compiles, loads and executes a typical program using C-shell file redirection is:

```
cft77 -es prog.f
segldr -o prog prog.o
prog < datax > & prog.outx
```

The source is on file *prog.f.* File *prog.l* is a source listing, which contains messages from the compiler, errors and vectorization information. File *prog* is the executable generated by the loader. File *datax* contains data read from unit 5. File *prog.outx* contains output from the program that is written to unit 6 and run time errors.

A generic C-shell script to perform the same tasks, with parameter substitution, follows:

```
#!/bin/csh
cft77 -es $1.f
segldr -o $1 $1.o
$1 < $2 > & $1.outx
```

If the above script is on a file *runit,* then the command

```
runit prog datax
```

executes the same commands as the first example.

3. PASCAL AND C ON THE CRAYS [c]

PASCAL, Standard C and Portable C compilers are supported on **Voyager** and **Sabre.**

3.1. PASCAL [c]

The CRAY PASCAL compiler is called *pascal.* Document CR-21, *CRAY Pascal Reference Manual* describes the CRAY implementation of PASCAL. Chapter 3 of CR-21 lists reserved words, operators and predefined identifiers. It complies with the Level 1 requirements of standard ISO 7185, defined by the International Organization for Standardization (ISO). The default compiler options may be explicitly overwritten by the command line. Compiler directives placed in a PASCAL program override the command line and default settings. CRAY extensions to PASCAL include:

1. Three sizes of integer data (24, 32 and 64 bit).
2. An OTHERWISE label for the CASE statement.
3. Calls to external PASCAL or FORTRAN routines.
4. The ability to initialize variables at compile time.
5. Array processing with a single statement.
6. In-line function expansion.
7. Octal numbers.
8. Expressions that operate on entire arrays.
9. Additional predefined declarations, functions and procedures.

The basic syntax of the *pascal* command line (All parameters are optional.) is:

pascal [-l file.l] [-b file.o] [-o olist] [-V] [file.p]

A complete description of the command line may be found in chapter 2 of CR-21. The parameters are:

-l file.l	Specifies the file to receive the listing output. All list output is suppressed if *-l 0* (zero) is specified. **The default is stdout.**
-b file.o	Specifies the file to receive the binary load modules generated by the compiler. **The default is file.o, unless the Input file is stdin, which default to stdin.o.**
-o olist	Specifies the list of compiler options, separated by commas, in effect at the beginning of compilation. **The defaults on the CRAY-2 are: a+, ag-, al-, bp-, breg=8, bt-, c-, cal-, debug-, dm0, e+, fe-, g-, l-, m2, ml3, o+, p-, p32, r+, rv-, st-, t+, treg=8, u-, v+, w-, x-, z+.** **The defaults on the CRAY Y-MP are: a+, ag-, al-, bp-, breg=8, bt-, c-, cal-, debug-, dm0, e+, fe-, g-, l-, ml3, o+, p-, p32, r+, rv-, st-, t+, treg=8, u-, v+, w-, x-, z+.**

-V	Writes the compiler version used to stderr.
file.p	Specifies the PASCAL source code file. It must end with the .p extension, unless the default is used. **The default is stdin.**

To compile, load and execute a PASCAL program with source on file, *test.p,* use:

```
pascal -l test.l -b test.o test.p
segldr -o test test.o
test > test.data
```

A compiler listing, *test.l,* is generated. The load modules are on *test.o,* and the executable is called *test.* When *test* is executed, the output is redirected to *test.data.*

With the 4.2 release of CRAY PASCAL, the ranges of I24 and I32 data types has expanded. I24 now spans -8388608 to 8838607 and I32 now spans -2147483648 to 2147483647. There are also new directives for inlining, producing CAL assembler code and aggressive optimization. User messages now conform to the UNICOS 6.1 message system conventions.

3.2. Mixing C and FORTRAN [d]

When a FORTRAN subroutine calls a C module, the C module names must be declared using upper CASE characters. Labeled COMMON names to link with FORTRAN routines must also be declared using upper case characters. Without these upper case declarations, the link phase will produce unsatisfied external references. Chapters 12 and 13 of *CRAY Standard C Programmer's Reference Manual,* which describe interlanguage communication and interfaces to libraries and the loaded, may be a valuable source of information.

3.3. Standard C [c]

CRAY supports the Standard C compiler, called *cc*. The *cc* compiler is described in **The C Programming Language Version 2** by Brian Kernighan and Dennis Ritchie. The reference manual for the CRAY implementation of Standard C is Document CR-30. There is also a *CRAY Standard C Library Reference Manual* (CR-31). The *cc* compiler is a superset of the *pcc* compiler (See section 3.3). Differences include:

1. Function prototypes, which provide a new way to declare functions.
2. Several new types, including long double, which is equivalent to FORTRAN DOUBLE PRECISION.
3. There are three new keywords: Const, signed and volatile.
4. Several new header files are included. The most significant for numerical processing is *<float.h>*.
5. Vectorization is enhanced.

The syntax for the use of the Standard C compiler is:

cc [options] files

The *files* usually have the extensions *.c*, *.o* or *.s*, which are Standard C source code, previously compiled code and assembly language code respectively. A file with the extension *.f* or *.F* is considered a *cft77* program, the latter to be run through the Standard C preprocessor prior to being compiled. A partial list of *options* follows:

-c	Forces the production of object files, and leaves them in the *.o* extension. The loader is not invoked.
-F	Enables flowtrace processing.
-g	Generates the Debug Symbol Table and suppresses any compiler optimization.
-h	Passes Standard C compiler code generation options to the Standard C compiler. These include optimization levels. See the *man* page for *cc* for more details.
-o	Specify a name for the executable code, i.e. *-o prog*. **The default name is a.out.**

Many other options are available, check the *man* entry for *cc*, which both compiles and loads Standard C programs.

3.4. Portable C [c]

CRAY also supports the portable C compiler called *pcc*. The *pcc* compiler is the portable C compiler described in **The C Programming Language** by Brian Kernighan and Dennis Ritchie. The reference manual for this CRAY implementation of the C programming language is Document CR-20. There is also a *CRAY C Library Reference Manual* (CR-28). The *pcc* compiler will not be supported after UNICOS 6. This implementation has several CRAY extensions, including:

1. Support of upper and lower case variables up to 255 characters.
2. Ability to pass a variable length argument list.
3. Support for declaration of more register variables.
4. Standardized frame package and stack-handling mechanism.
5. Support for enumeration and unique structure member names.

The syntax for the use of the portable C compiler is:

pcc [options] files

The *files* usually have the extensions *.c*, *.o* or *.s*, which are C source code, previously compiled code and assembly language code respectively. A file with the extension *.f* or *.F* is considered a *cft77* FORTRAN program, the latter to be run through the C preprocessor prior to being compiled. A partial list of *options* follows:

-c	Forces the production of object files, and leaves them in the *.o* extension. The loader is not invoked.
-F	Enables flowtrace processing.
-g	Generates the Debug Symbol Table and suppresses any compiler optimization.
-h	Passes C compiler code generation options to the C compiler. These include optimization levels. See the *man* page for *cc* for more details.
-o	Specify a name for the executable code, i.e. *-o prog*. **The default name is a.out.**

Several other options are available, check the *man* entry for *pcc,* which both compiles and loads portable C programs.

4. UNICOS COMPUTING ENVIRONMENT [b]

UNICOS is designed primarily as an interactive system. The user has the ability to customize his environment and to create new commands (or scripts) to perform frequently executed tasks. The environment can be created automatically at login by using the *.login* and *.cshrc* files (See sections 4.1.4 and 4.1.5) or as needed by modifying the environment or C-shell variables (See section 4.1.2). Usually, the *.login* file is used to initialize batch vs interactive environment, your search path for commands and environment variables. The *.cshrc* file is used to insure that all your aliases are passed on to subsequent C-shells and to initialize C-shell variables.

4.1. Logging Onto the CRAY Supercomputers [c]

When you get connected to **Voyager** the system responds

```
Connected to voyager.larc.nasa.gov
Escape character is '^]'

Cray UNICOS (voyager)

UNICOS Release 6.1
```

A similar message appears when you get connected to **Sabre.** The system then prompts for your login name with:

```
login:
```

At this time type your login name followed by a carriage return. The carriage return is followed by a prompt for your password:

```
password:
```

If you type either your login name or password incorrectly, the system prompts you again. If you hit the backspace in an attempt to correct an error, your login attempt fails. Try again and be more careful. If you can't login at all, call Password Validation at 864-8282. If your login is successful, you are told the name of your default account (See section 1.3.1 of A-8).

4.1.1. Changing Your Password [c]

You must change your initial password when you first login to both **Voyager** and **Sabre. When you first login to Voyager or Sabre, you are told to change your password immediately. Once you successfully make the change, the system closes your connection. Don't panic, just login again with the new password.** You must change your password at least once a year. Once changed, a password may not be changed during the next week. Your new password must meet the following security requirements:

1. It must be at least six characters long.
2. It must have at least two alphabetic and one numeric or special character.
3. It must not be any permutation of your login name.
4. It must differ from your old password by at least three characters.

Also, words found in a dictionary with only a single digit appended to the end or added at the beginning to form the password are highly susceptible to being compromised and should not be used. The command to change your password is:

passwd

It is an interactive command. All that you must do is enter the command and *passwd* prompts you for a response. As you enter your old and new passwords at the appropriate prompts, you will notice that the system does not echo your password to the screen. This is a security feature and the reason that the system prompts you twice for your new password. If the two entries for your new password don't match, the system does not change your password and prompts you again to enter your new password. If you forget your password, contact Password Validation at 864-8282 for assistance.

4.1.2. Environment and C-Shell Variables [b]

Data about your environment such as your home directory and terminal type is maintained in two sets of variables called environment and C-shell variables. Environment variable values are inherited by all programs executed by the shell, including new shells that you spawn or fork. C-shell variables are inherited by execution of your *.cshrc* file whenever you spawn a new C-shell. Many of your environment and C-shell variables are defined by the system administrator in your initial *.cshrc* and *.login* files, but you may change these variables by using the *setenv* or *set* commands respectively. These variables generally define information that programs need to execute correctly. Your current environment variables can be displayed by typing:

```
printenv
```

The *setenv* and *set* commands have slightly different syntax, as illustrated below

```
setenv variable string
```

and

```
set variable=string
```

where *variable* is the name of the environment variable and *string* is the new value. Some of the C-shell variables are Boolean and are either *set* (i.e. true) or *unset* (i.e. false), such as *noclobber* and *ignoreeof.* These two C-shell variables are used in the sample *.cshrc* file in section 4.1.5.

Some common environment variables include HOME, PATH, TERM, USER, DISPLAY, SHELL, PRINTER, DELIVER and EDITOR. Environment variables are discussed in detail in section 5.3.1 of CR-2. Some will be illustrated in the examples of *.login* and *.cshrc* files that are discussed in sections 4.1.4 and 4.1.5.

4.1.3. Personalizing Your Environment [a]

A nice feature of UNICOS is the ease with which you can change your environment. It can be done automatically with the *.cshrc* and *.login* files, or it can be custom tailored for a single interactive session. When you are assigned a login name and password, the system administrator provides default *.cshrc* and *.login* files in your home directory. You may change these to suit your needs. A frequent change that users make to the *.cshrc* file is to add aliases. Aliases allow you to create another name for frequently used commands. For example if you type (or add to your *.cshrc* file) the next line

```
alias rm "rm -i"
```

then each time you type *rm* to delete a file the system automatically prompts you to insure that you really intended to remove the specified file. Otherwise you would have to enter

```
rm -i
```

each time you wanted the system to prompt you for file removal.

The directory */usr/local/adm/skel* has several sample hidden files that you may use to customize your environment, including *.login, .cshrc* and *.logout* files. Additionally there are sample skeleton files for the *.exrc, .mailrc, .forward, .netrc* and *.rhosts* files that are discussed in Document A-8. The next two sections have sample *.login* and *.cshrc* files that are similar to the skeleton files that the system administrator gives you initially. If you modify the default files or bring your *.login* and .cshrc files from another machine and discover that something isn't working, then you can copy the default *.login* and *.cshrc* files from the directory */usr/local/adm/skel* to your home directory.

4.1.4. Your .login File [d]

The *.login* file is automatically executed every time that you log into either CRAY after your *.cshrc* file is executed (See section 4.1.5). An analogous *.logout* file is automatically executed every time that you log off, if such a file exists (See section 4.2.3). An example of a typical *.login* file is given below. It is similar, but not identical to the skeleton *.login* file that you are given initially. Everything to the right of the pound signs (#) is a comment to explain the various entries. Your *.login* file may not look exactly like this, but you can tailor it to suit your needs.

Sample .login File

```
set path=(~/bin /usr/local/bin /usr/local /bin /usr/bin /usr/ucb .)  # Set path variable
echo "The current date and time are: "                              #
date                                                                 # Give date and time at login
umask 022                                                            # Deny write access to group
                                                                     #  and others
setenv DELIVER delivery_info                                         # delivery_info - 8 chars max
source /usr/local/adm/skel/GRAPHICS                                  # For access to di3000
if ($?ENVIRONMENT) then                                              # Check for batch or interactive
  if ("$ENVIRONMENT" == "BATCH") then                                #  environment
     exit                                                            #
  endif                                                              #
endif                                                                #
echo "Enter Terminal Type -                                          #
   Default is vt100 "                                                # Check terminal type
set termname="$<"                                                    #
if ($termname =~ ) then                                              #
   set termname=vt100                                                #
endif                                                                #
setenv TERM `tset -Q - $termname`                                    #
set term=$termname                                                   #
quotamon -s 600                                                      # Ask for automatic warning
                                                                     # when soft quota is exceeded
mesg -n                                                              # Disable talk utility
setenv MAIL /usr/mail/$user                                          # Identify mailbox
setenv PAGER /usr/ucb/more                                           # One screen of mail at time
stty erase ^H kill ^U                                                # Set erase & line kill chars
setenv EXINT "set redraw wm=8 showinsert ai"                         # Set vi attributes
setenv VISUAL /usr/bin/vi                                            # Use vi for ~v in mail
setenv EDITOR /usr/bin/vi                                            # Use vi for ~e in mail
setenv LPP 48                                                        # Set default page length
setenv DISPLAY myxterm:0:0                                           # Define display for X
```

4.1.5. Your .cshrc File [c]

Every time you log onto either CRAY or spawn a new C-shell the *.cshrc* file is automatically executed. It may also be executed by typing:

```
source .cshrc
```

An example of a typical *.cshrc* file is given below. Everything to the right of the pound signs (#) is a comment to explain the various entries. Your *.cshrc* file may not look exactly like this one, but remember that you can tailor it to suit your needs.

Sample .cshrc File

```
alias cds "cd /scr/$user"                # Shortcut to scratch directory
set cdpath=(~ )                          # Set alias for default cd
if ($?ENVIRONMENT) then                  # Check for interactive or batch
  if ("$ENVIRONMENT" == "BATCH") then    #  environment.
    exit                                 #  Following commands are
  endif                                  #  only executed for
endif                                    #   interactive shells
set history=20                           # Save last 20 commands on history file
set savehist=20                          # Save last 20 commands between logins
set prompt="$user% "                     # Set prompt to show user
set ignoreeof                            # Disable ^D for logout
set noclobber                            # Avoid accidental file overwrite
set notify                               # Automatic notification of mail
                                         #  and background task completion
alias mail mailx                         # Always use smart mail
alias h history                          # Shorthand notation for history
alias c clear                            # Clear the screen
alias rm "rm -i"                         # Avoid accidental removal of files
alias ls "ls -aC"                        # List files in multiple columns
                                         #  including "." files
alias x "chmod +x"                       # Make a file executable with
                                         #  "x file_name"
alias bye logout                         # Alias for logout
```

More information on the commands listed above or in the sample *.login* file can be found using the *man* command for the specific command or with

```
man csh
```

Section 5.3.2 of *The CRAY UNICOS Primer* has more detail on the *.login* and *.cshrc* files.

4.2. Interactive Computing

UNICOS is basically an interactive operating system, but unlike some other operating systems the user can be executing multiple tasks, called processes under UNICOS, at the same time. This feature tends to make your terminal sessions more productive, since you don't have to wait for one process to finish to start another.

4.2.1. Processes [a]

A process is a program that is running. The most frequently running program is the command interpreter called *csh* and referred to as the C-shell. UNICOS also supports the Bourne shell, which is called *sh.* Every time the user issues a command the *csh* spawns (or forks) a new process. The spawned process is the child of the process that created it. Processes may be run in the foreground or background. Usually they are run in the foreground; however by running processes in the background, you are able to do a lengthy task, while continuing to do other work in the foreground. For example, to compile a FORTRAN program, prog.f, in the background type:

```
cft77 prog.f &
```

The ampersand (&) causes the program to be compiled in the background. If you are running in the background and logout, your process will terminate unless you have redirected standard input and output (See section 2.4.3) and have preceded the command with the *nohup* (no hang up) command. For example,

```
nohup cft77 prog.f &
```

continues to execute after you logout.

You can check on the status of processes with the *ps* command, which stands for process status. Each process is assigned a number called a Process IDentifier (PID). This number is important if you have a hung process and want to kill it (See section 4.2.2). It is not possible to switch jobs between the foreground and background on **Voyager** or **Sabre** as it is on the CONVEX computers.

4.2.2. Killing a Runaway Process

If you have a runaway process running in the background, the *ps* command gives you the PID which you can then use to kill the process. For example to kill the process with PID 1234, type:

```
kill -9 1234
```

Sometimes you may have a hung terminal or a runaway process running in the foreground. There is no need to panic. First try using the ^C (control C) signal. If that doesn't work, login again from another terminal and enter the *ps* command. Then use the *kill* command as just illustrated to kill the process(es) associated with the other terminal. If you kill the wrong C-shell by mistake, you are logged off the system, so just try again.

4.2.3. Logging Out

Usually, you will log off the system with the *logout* command. If you are no longer in your login shell then you must use the *exit* command. The ^D (Control D) will also log you off the computer if you do not have the variable *ignoreeof* set in your *.cshrc* file (See section 4.1.5). You may also create a *.logout* file that is executed every time that you logout. A sample *.logout* file is given below.

Sample .logout file

```
clear                                   # Clear screen at logout
/bin/rm a.out                           # Remove generic executables
echo "Closing connection to Sabre at "  #
date                                    # Give date and time at logout
```

4.3. Batch Processing on the CRAY Supercomputers [d]

The Network Queuing System (NQS) provides the batch processing environment on the CRAY-2S. It may be necessary to tune the various queue limits to the mix of jobs at LaRC to obtain the optimal throughput. The NQS queues for **Voyager** (as of March 1992) are found in **Table 4.1** at the end of the next section. Those for **Sabre** (as of November 1992) are found in **Table 4.2.** Changes to the NQS queues are announced in the Message-of-the-Day (See section 8.1.2 of A-8), referring you to the new entry under *notes CRadmin* (See section 8.3 of A-8) on **Eagle** and **Mustang.**

4.3.1. NQS [d]

NQS is summarized in section 6.2 of CR-2. The NQS commands are all described in CR-6 and on *man* pages. NQS commands include:

qsub	Submits a request to a batch queue.
qstat	Displays the status of NQS queue and requests.
qdel	Deletes or signals NQS request.
qlimit	Displays batch limits and shell strategy

To use NQS, you must first create a script (See section 2.1.3 of A-8) of commands to be executed in batch mode. Unlike interactive shell scripts, NQS scripts can not accept positional parameters. Since NQS begins execution in your home directory, file space limitations will generally require that you use the *cd* command as one of the first commands in your script to change into your subdirectory in */scr* (See section 4.4.1), for the compilation, loading and execution of your program. The NQS script may have a prolog that defines the shell under which it executes and sets resource limits on a per-process or per-request basis. By default, NQS scripts are executed in your login shell, which is the C-shell for nearly all **Sabre** and **Voyager** users.

The resource limits are specified with # *QSUB* as the first characters of the line. For compatibility with earlier systems, the string # *@$* may be used as the first characters of the line as well. Examples of frequently used *qsub* options are:

-lm 18Mw	Establish a per-process memory size limit of 18Mw.
-lt 1000	Establish a per-process CPU time limit of 1000 seconds.
-nr	Specify that the process is not rerunnable.
-me	Send mail when the request ends execution.
-q q8mw_1h	Queue request in the *q8mw_1h* queue.
-a 5pm	Submit job to NQS at 5 PM.

If *-lM* and *-lT* are used, the memory size and CPU time limits are on a per-request basis. Check the *man* page for details on these and other *qsub* options.

An example of a sample NQS script for user *john* follows:

```
#
# QSUB-lm 20mw
# QSUB-lt 1000
# QSUB-me
#
cd /scr/john
/bin/time cft77 -es prog.f >& prog.out
segldr -o prog prog.o
/bin/time prog < prog.in >& prog.out
```

This job when submitted to NQS, will run in the scratch directory of *john,* with a memory size limit of 20 million words and a CPU time limit of 1000 seconds. The job times both the compilation and execution (See section 6.1) and the request will send *john* a mail message when it completes.

The most common syntax of a batch job submission using NQS is

```
qsub script_name [-q queue_name]
```

where specification of the *queue_name* is optional, but the limits specified in the script must not exceed those of the specified queue. Once you submit the job, the system responds:

request *request_name* **submitted to queue:** *queue_name*

Unless you specify a specific queue with the *-q* option, the *queue_name* that will be given is *router.* A later invocation of the *qstat* command will show the specific queue to which the job was assigned. See **Table 4.1** and **Table 4.2** for the queue configurations. The *request_name* is used by the *qdel* command, if it becomes necessary to kill the request. To kill a request, use:

```
qdel -k request_name
```

The *-k* option is required only if the process has begun execution.

Once your NQS job is completed, two files are created in the directory from which you submitted the job. If your *script_name* is *nqsjob* and your *request_name* was *4758,* then these files are named *nqsjob.e4758* and *nqsjob.o4758.* They contain information that would have been written to standard error and standard output respectively. Both files should be examined for information if your program terminated abnormally.

The *qlimit* command displays the resources available on **Sabre** and **Voyager** and the corresponding parameter names to be used by *qsub.* The shell strategy displayed is the name of the shell which interprets script commands, if the user does not specify an alternate shell. The display is of the form

Per-Process/Request *resource_name resource_description*

where *resource_description* defines the *qsub* option associated with *resource_name.*

The *qstat* command is used to query the system about the status of NQS requests as illustrated below:

qstat -a	Show summary of all requests.
qstat -i	Show summary of requests awaiting processing.
qstat -r	Show summary of requests currently running.
qstat -b	Show batch queue summary.

The status of an individual queue may also be queried. For example,

qstat -rbi big-long

gives a summary of all jobs running or queued in the *big-long* queue, which enables you to check on the status of your job. Running jobs are listed first, followed by queued jobs which are listed in the order that they become eligible for execution. The *qstat* command has other options, check the *man* page.

NQS Queue	Maximum Memory (Mw)	CPU Time (secs)
interactive *	10	600
q8mw_1200s	8	1200
q8mw_1h	8	3600
q8mw_2h	8	7200
q20mw_1h	20	3600
q20mw_3h	20	10800
db20mw_100s	20	100
q40mw_2h **	40	7200
q40mw_4h **	40	14400
db40mw_300s	40	300
multitask	40	600
special	110	14400

* *Interactive is not a true NQS queue, but is included for completeness.*

** *Queue is active only during non-prime shifts.*

Table 4.1 NQS Queues for Voyager [c]

These NQS queues were established March 4, 1992. This change only affected the names of the queues to reflect the time and memory restrictions of each queue. They may change, so space has been left in **Table 4.1** for additional entries, or changes to the queue limits. Jobs are automatically routed to the correct NQS queue as specified by the *-lm* and *-lt QSUB* options, except for *special* and *multitask* which must be specified with the *-q* option on the *qsub* command. The *notes CRadmin* category on **Eagle** and **Mustang** contains the latest revision to **Table 4.1.** The *special* queue is reserved for priority jobs. If you need this capability, contact Joe Drozdowski at (804)-864-6535. The *multitask* queue is only enabled a limited number of times per week.

WARNING: Any job submitted to NQS that exceeds the established queue limits will be deleted by the NQS daemon and will never appear with a *qstat* command. However, the daemon does send you electronic mail to inform you that your request could not be queued.

NQS Queue	Maximum Memory (Mw)	CPU Time (secs)
interactive *	10	600
q10mw_1h	10	3600
q10mw_3h	10	10800
q10mw_6h	10	21600
db20mw_100s	20	100
q20mw_1h	20	3600
q20mw_3h	20	10800
db40mw_300s	40	300
q40mw_1h	40	3600
q40mw_3h	40	10800
q80mw_3h	80	10800
db125mw_300s	125	300
q125mw_3h	125	10800
mt40mw_3h	40	10800
mt200mw_8h	200	28800
ssddb40mw_300s	40	300
ssd40mw_3h	40	10800
ssdmt40mw_3h	40	10800
ssdmt200mw_8h	200	28800
special	225	32400

** Interactive is not a true NQS queue, but is included for completeness.*

*** Queue is active only during non-prime shifts.*

Table 4.2 NQS Queues for Sabre [d]

These NQS queues were established November 16, 1992. They may change, so space has been left in **Table 4.2** for additional entries, or changes to the queue limits. See Section 4.4.1 for information about the special queues for using SSD. Jobs are automatically routed to the correct NQS queue as specified by the *-lm* and *-lt QSUB* options, except for *special* and *multitask* which must be specified with the *-q* option on the *qsub* command. The *notes CRadmin* category on **Eagle** and **Mustang** contains the latest revision to **Table 4.2.** The *special* queue is reserved for priority jobs. If you need this capability, contact Joe Drozdowski at (804)-864-6535.

4.3.1.1. NQS Queue Management [d]

The following management strategy has been adopted to insure each user an equitable portion of resources on **Sabre** and **Voyager** when using NQS. Changes to this strategy are announced in the Message-of-the-Day and under *notes CRadmin* on **Eagle** and **Mustang.**

1. No user may have more than one job (queued or running) in any queue Monday through Thursday, with a limit of jobs in 3 queues. On Friday, starting at 8 A.M., users may submit up to four jobs per queue to be run during the weekend. Users should, however, be reasonable with the number of jobs submitted for weekend processing.
2. The debug queues are intended only for debugging. They are not for short production runs. The limit in these queues is one job per user in any state. Running many jobs in these queues tends to cause swapping problems. Their use is expected to be minimal. Running more than 4 or 5 jobs each day for an extended period is considered an abuse. It prevents other users from preparing jobs for the larger queues as the queues were intended.
3. The emphasis on **Sabre** is running large memory jobs, but not at the expense of oversubscribing memory and causing swapping. The use of the Solid-state Storage Device (SSD) is limited to special queues, which require that users ask for validation to use the SSD.

Failure to comply with the above guidelines results in all NQS jobs for the offending user being put into a hold status. The user is notified by mail of this action. When the user complies with the guidelines, remaining jobs have the hold removed. A hold does not allow a job to be considered for selection to run.

It is requested you be reasonable with your weekend requests or similar guidelines may be forthcoming. It is your responsibility to comply with the guidelines on Monday, if you stacked jobs on Friday, Saturday or Sunday.

4.4. Solid-state Storage Device (SSD) [d]

The CRAY Y-MP, **Sabre,** configuration includes a 512 million word Solid-state Storage Device (SSD). The SSD functions like a disk, but enhances the performance of the CRAY Y-MP because of SSD's fast access time, fast transfer rates, and large storage capacity. Three hundred sixty million words of SSD are available to the users.

Initially, use of the SSD is through the batch system NQS only. The batch queues defined below are used to manage the file space allocation so that competing jobs do not oversubscribe the SSD and cause other executing jobs to abort. The Session Reservable File System (SRFS) software from NAS will be implemented as soon as possible to provide a more automated management mechanism.

4.4.1. SSD Queue Structure and Validation [d]

Four queues have been established to allow the use of SSD. The queues which allow use of SSD are *ssddb40mw_300s, ssd40mw_3h, ssdmt40mw_3h,* and *ssdmt200mw_8h.* A user must be validated to use these queues. To become validated, e-mail your request to be added to the SSD user list to *root@sabre.* Each validated user has a quota limit of 180 million words of SSD.

4.4.2. SSD Job Submission [d]

To submit a job for a single cpu and SSD use:

```
qsub -q ssd  scriptname
```

The explicit option, *-q ssd,* on the *qsub* command or the equivalent embedded option in the script will cause the job to be routed to either the *ssddb40mw_300s* or the *ssd40mw_3h* queue.

To submit a job for multitasking and SSD use:

```
qsub -q mtssd  scriptname
```

The explicit option, *-q mtssd,* on the *qsub* command or the embedded equivalent option in the script will cause the job to be routed to either the *ssdmt40mw_3h* or the *ssdmt200mw_8h* queue.

4.4.3. SSD File Cleanup [d]

Files residing on SSD remain there until some action is taken to move or delete them. If no action is taken, they reduce the space available for subsequent jobs. Consequently, SSD files must be moved or deleted by the user. Any files left there after job completion, or files observed to have been generated from a job not running from one of the valid SSD queues will be removed immediately. Therefore, at the completion of your job, be sure to :

1. Move any files which need to be saved to permanent or mass storage.
2. Remove all files from SSD space.

4.4.4. SSD Sample Script [d]

The SSD file system can be accessed with the preset environment variable *$FASTDIR*. The following sample script, *runssd,* uses embedded NQS options. It assigns the file *plf* (to be opened as FORTRAN unit 40) to SSD using the unblocked structure required for the most efficient transfer rates, executes the job and removes the files from SSD.

```
#
#QSUB-lt 10800
#QSUB-lm 40mw
#QSUB -q ssd
#
cd /scr/userx
cf77 prog.f
/bin/rm asyn
env FILENV=asyn assign -a $FASTDIR/plf -s u u:40
env FILENV=asyn a.out < data >& out
cp plf ~userx/plf
/bin/rm $FASTDIR/plf
```

The *assign* command associates attributes with unit numbers or file names during the processing of a FORTRAN **OPEN** statement. These attributes are stored in an assign environment file called *asyn* in the above example. Then the executable is executed with the file attributes stored in *asyn*. See the *assign* man page for more details.

To submit the script:

```
qsub runssd
```

The job will be put in the *ssd40mw_3h* queue.

5. DEBUGGING ON THE CRAYS [c]

The symbolic debugger, *cdbx,* supports most of the features of *dda,* the dynamic dump analyzer and *drd,* the dynamic run-time debugger, as well as standard features of the UNIX debugger *dbx.* Neither *dda* or *drd* is supported under UNICOS 6.1.6. The *cdbx* debugger is described in CR-25, *CRAY UNICOS CDBX Debugger User's Guide,* and CR-26, *CRAY UNICOS CDBX Debugger Reference Manual.*

Some debugging utilities require the file *core,* which contains a complete image of the failed program. Because **Voyager** and **Sabre** have large memories, these files can become huge. Under UNICOS 6.1, on both **Voyager** and **Sabre,** a full length *core* file is always obtained when a program terminates abnormally. On **Voyager,** most run-time errors did not create a *core* file under UNICOS 5.1, simply to conserve file space.

Another alternative is to use the traceback mechanism to find errors. There is a FORTRAN callable routine TRACEBK that permits dynamic tracebacks from an executing program. There is also a locally written script that takes an address from an aborted subroutine and returns a FORTRAN line number to assist in debugging (See section 5.4). Next, there is a another locally written script, *debugx,* that executes your programs and invokes the CRAY *debug* utility if your program terminates abnormally (See section 5.5). Section 5.6 has been added to provide information about numbered FORTRAN run-time errors and to describe some unnumbered miscellaneous error messages that are otherwise difficult to decipher.

5.1. cdbx Symbolic Debugger [b]

The *cdbx* debugger is an interactive symbolic debugger that has the capability to perform the following functions:

1. Setting breakpoints and traces.
2. Controlling program execution.
3. Displaying and changing data.
4. Managing image (core or executable) files.
5. Defining debug variables and aliases.

Multitasked applications can also be debugged using *cdbx,* but only by examining the *core* file. Interactive debugging of multitasked programs is not yet supported. Chapter 4 of CR-25 has sample debug sessions, while chapter 3 summarizes the *cdbx* commands, which are organized by functionality, while chapter 7 of CR-26 describes the *cdbx* commands in detail. Almost all the *dbx* commands are implemented with this release of *cdbx.* Additionally, several CRAY-specific commands have been included. The *cdbx* utility is very similar to the CONVEX utility *csd.* Error messages are explained in chapter 4 of CR-25.

To use *cdbx,* it is recommended that each source code module be on a separate file. Using optimization inhibits the effectiveness of the debug tables that are generated with the *-ez* compiler option. The best results from using *cdbx* are with unoptimized code. You should use the *-o off* compiler option on routines where you suspect errors, otherwise you may have difficulty setting breakpoints and traps at desired locations within your code.

This release of *cdbx* supports an X Window System (X) interface in addition to the traditional line oriented interface. If you are invoking *cdbx* from within the X environment, and have the *DISPLAY* environment set, then the X interface to *cdbx* is invoked unless the *-L* option is selected (See section 3.4 of A-8). It also allows the command *dbx* to invoke *cdbx,* along with a warning message that UNICOS does not support *dbx.* Appendix B of CR-26 describes differences between the current implementation of *cdbx* and *dbx.* To use *cdbx,* symbol tables must have been generated when the program was compiled. Section 2.1 describes how to generate the symbol tables with each of the compilers supported on **Voyager** and **Sabre.** A simplified syntax of the *cdbx* command follows on the next page.

cdbx [-c cfile] [-e efile] [-l lfile] [-s symfile] [command_line]

-c cfile	select the file that contains a dump of the user memory image. **The default is core.**
-e efile	enable the echo feature to record the input of the debugging session on efile. **The default file name is echo.db**
-l lfile	select the logging feature and identify the file to which all log output is written. **The default is log.db**
-s symfile	select the file that contains the symbolic information. **The default is a.out.**
-L	forces use of the line oriented interface rather than the X interface
command_line	specify the command line that is used to invoke the program, including all options and file redirection. The command line must be enclosed in double quotes (" "), unless it consists only of an executable name. **You must include blank space around the " < " and " > " symbols used for file redirection in order for *cdbx* to recognize your command_line.**

Other *cdbx* options are available, including overriding the *DISPLAY* environment variable with the *-display* option, check the *man* page. By default, *cdbx* searches your current working directory for the file *a.out* to obtain the necessary symbolic information. The *-s symfile* option may be used to override the default. It also searches for a *core* file, which may be overridden using the *-c cfile* option. If a *core* file exists, it is used as the initial image to be debugged, otherwise the executable file is used as the initial image to debug. See Chapter 5 of CR-26 for more detailed information on invoking *cdbx.*

5.2. symdump Snapshot Dump

The snapshot dump utility, *symdump,* yields output similar to that of *debug* (See section 5.3), but can be invoked repeatedly by an executing program with a subroutine CALL. To use *symdump* from within a FORTRAN program, insert the following line of code at all locations where a snapshot dump is desired:

CALL SYMDUMP (["options"])

The *options* argument is a character string containing *debug* options. The *symdump* utility is described in chapter 5 of CR-10.

5.3. debug [c]

The *debug* utility (See chapter 4 of CR-10) is a batch symbolic post-mortem core analyzer that allows you to interpret a *core* file in terms of source language symbols, which are user-defined names for variables and subroutines. It also provides a subroutine traceback and can be used with either the *cft77* or *cft* compiler. To use *debug,* your program must be compiled with the *-ez* option for *cft77* or the *-ezi* option for *cft.* It is recommended that optimization be turned off when you attempt to use *debug,* otherwise it will not be able to give the correct loop indices where the error occurred. A *core* file must exist to use *debug.* The syntax of *debug* is:

debug [-B] [-b blklist] [-d dimlist] [-s symfile] [-Y] [-y symlist]

There are four ways to display COMMON blocks and program variables. If neither -*B* or -*b* is specified, then no COMMON blocks are displayed. If both are specified then only those COMMON blocks named are displayed. Just specifying -*b* displays only the COMMON blocks named and using just -*B* displays all COMMON blocks. However, if neither -*Y* or -*y* is specified, all variables are displayed. If both are specified then only those variables named are displayed. Specifying just -*Y* displays no variables, while specifying -*y* displays all variables except those named.

The file *symfile* contains the debug symbol tables. The default is *a.out.* The parameter *dimlist* specifies the maximum number of elements to be displayed for array variables. The default is 20,5,2,1,1,1.

5.4. Interpreting the Error Location in a CRAY Dump [d]

Using *cft77,* the traceback from a floating point exception run-time error does not give a FORTRAN line for the subroutine in which the error occurred. The script *~howser/err.exc* on **Voyager** and **Sabre** may help you find a FORTRAN line number near the error. The use of the utility requires a traceback like the one given on **Voyager,** but this is not the default traceback on **Sabre.** On **Sabre** only, to get a traceback which can be used by this script, issue the command,

setenv $TRACEBK2 1

before executing the job. The environment variable, *$TRACEBK2,* can be set in either the *.cshrc* or *.login* file.

The script requires two parameters that are obtained from the dump information given at abort time. The line containing the information required is of the following form:

```
"tlns" FltPt exception: CPU-A stopped at P= 05324c = "TURB" + 2477c

A0 ...          S0 ...
A1 ...          S1 ...
 .
 .
```

In the information above:

TURB = routine in which error occurred
2477c = relative address where error occurred

The script is executed as follows:

~howser/err.exc subname addr

subname The name of the routine in which the error occurred. It must be a separate *.f* file containing only *subname* and is compiled with all the *cft77* defaults.

addr Relative address where error occurred. **(Do NOT include the alphabetic ending.)**

The script only requires the cpu time needed for a *cft77* compilation of the routine. The output goes to *stdout* and returns up to four line numbers that will be near the error. The script will be no more accurate than the address returned with the abort dump. With optimization and vectorization this can be several lines away from the error, but may point to the loop containing the error.

Using the sample abort above and assuming that the subroutine TURB is on the file *turb.f,* the script is executed:

```
~howser/err.exc turb 2477 > err.out &
```

The file *err.out* contains the FORTRAN line number(s) near the error.

5.5 Automatic Symbolic Dump [c]

When your executing job terminates abnormally, UNICOS usually provides insufficient information on the problem for the error to be located. In this situation, you are forced to rerun the job to perform any debugging. A low overhead script has been developed that forces a symbolic interpretive dump of program variables at an abnormal termination.

The script is called *debugx* and is found in the directory *~howser/Debugx* on **Voyager** and **Sabre.** To use *debugx,* your program must be compiled with the *-ez* option to build a symbol table. The utility executes your program and calls the CRAY utility *debug* (See section 5.3) only if the program terminates abnormally. The only overhead associated with *debugx* when your program terminates normally, is a minimal increase in compile time to build the symbol tables. Thus the script may be used on a regular basis.

To use the script, your normal execute line must be enclosed in double quotes (" ") to prevent the shell from trying to interpret the command before *debugx* can. For example, if your code is on a file *prog.f,* the following sequence of commands may be used to compile, load and execute your program under the control of *debugx:*

```
cft77 -ez prog.f
segldr -o prog prog.o
~howser/Debugx/debugx "prog < mydata > & outdd"
```

The symbolic dump is appended to the file *outdd.* This information should always be redirected, since the file can be very large. The most recent values of program variables known to debug for the subroutine in which the error occurred, as well as values for variables for all routines in the calling tree are included. As with any debugger, the use of optimization or vectorization may mask the true location of the error.

If you have difficulty interpreting the results from *debugx,* call Lona Howser at 864-5784 for assistance. You may use the *ln* command to link to *debugx* or may copy it to your own directory. There is a *README* file in the directory *~howser/Debugx* that gives more details about the script and the form of the output.

5.6 Error Messages [d]

Under UNICOS 6, an attempt has been made to consolidate the error messages issued by the various utilities in the *libf* FORTRAN run-time library for both **Voyager** and **Sabre.** In the past error number 121, for example, might be completely different on the two architectures. The *CRAY UNICOS FORTRAN Library Reference Manual,* CR-27, has a detailed list of all errors issued by *libf* .utilities, as well as a brief description of the condition that may have caused the error. Messages now begin at 1000 rather than 100 to prevent misinterpretation with the UNICOS 5 messages.

5.6.1 Numbered Error Messages [c]

The UNICOS command, *explain,* is a useful in interpreting runtime errors issued by the FORTRAN run-time library and some other utilities. Many run-time errors are given with a message identifier and an error number. To receive an explanation of the error message and a description of what conditions may have caused it, execute the command *explain* with the message identifier and error number as parameters. For example, if your program received the message:

lib-1315 : UNRECOVERABLE library error

type:

explain lib-1315

and more information about the possible causes of this error will be given.

5.6.2 Unnumbered Error Messages [d]

These following unnumbered error messages are difficult to decipher, unless you have encountered them in the past. The first message may appear on **Voyager** only

Run-time stack overflow: Memory request denied by UNICOS
LA = 03155537(8)

If you get this message, ask yourself the following questions:

1. Are you using the old cft compiler?
2. Are you running a program using *cft* instead of *cf77*?

The loader does not accurately reflect the memory needed if a big array is in a dimension statement, but not in labeled COMMON . The *cft77* generated code seems to load ok, but the old *cft* compiler does not compute the accurate sizes needed. One of the following actions usually cures this problem.

1. Put the big dimension array variable in a labeled common and the memory, stack, etc. will be correctly evaluated and size will give you, a more accurate memory neeeded.
2. Increase the *-lm* parameter for your NQS job to reflect the memory really needed.
3. If you were running interactively, run the job under NQS. Be sure the *-lm* parameter gives enough memory. It actually may be needing more than the 10 million words, but gives this message instead of "not enough space".

The next message also is frequently seen on **Voyager** only.

> **Local memory usage overflowed available space by nnn words;**
> **local block overlay performed**

This is a caution message issued by *segldr.* It may occur when you have compiled a big source code (100 or more subroutines). A big source code is generated if the array bounds checker option is enabled. The local memory on **Voyager** overflowed, but *segldr* was successful at overlaying the program. Your results should be fine. The next error messages may be encountered on both **Voyager** and **Sabre.**

> **USER CALLED PREDEFINED PROCEDURE HALT**
> **Runtime error RT1000: program called HALT**
> **Error RT1000 is fatal**
> **Detected by ROR[145] Write error.**

Any of these messages come from the compiler and usually mean:

1. You are compiling in your home directory and have exceeded your permanent file quota. Type the command *quota* and see if you have reached your limit. Compile the job in */scr.*
2. If you are compiling in scratch, */scr* may be full. This occurs intermittenly, because it varies with what other users are doing. Type the command *df /scr* to see how much space is left.

> **Not enough space**
> **Cannot allocate memory space**

These two messages may be received from an interactive or a batch job. Either message means the executable requires more memory than allowed by the queue requested or needs more memory than the allowed interactive limit. Most of the time the message "Not enough space" will be issued immediately and the job never gets into execution. Sometimes the job will be close to the memory limits and will get into execution, but still not have enough memory, then the message "Cannot allocate memory space" will be issued. For either message the solution is to run the job in a queue which allows more memory. If the job is an interactive job, the solution is to run it as a batch job in a queue with sufficient memory. If the job is a batch job, the solution is to increase the memory size requested and the job will be put in a larger queue.

6. CODE MANAGEMENT AND CONVERSION

The FORTRAN compiler, *cft77,* is based on standard FORTRAN 77, but does have some extensions and restrictions that are unique to CRAY supercomputers (See section 6.2). There is no explicit vector syntax, so programs written in standard FORTRAN-77 should need little work to execute correctly on **Voyager** or **Sabre.** To get the optimal performance from your program is slightly more difficult than just obtaining the correct answers (See section 6.2).

6.1. Timing Your Program [c]

One of the first things that you may want to do once your program is executing correctly is to time it. You can use the */bin/time* command to time your compilation and execution in the shell script that runs your program, as follows:

```
/bin/time cft77 prog.f
segldr -o prog prog.o
/bin/time prog < prog.in > & prog.out
```

The */bin/time* command returns elapsed time, CPU time and system time labeled as *real, user* and *sys* respectively. The C-shell has a built-in *time* command that returns timing information in a different format (i.e. *user, system, real* and ratio of *user* and *system* time to *real* time expressed as a percentage). Timing multitasked codes is described in section 6.3.6. The FORTRAN compiler *cft77* is discussed in section 2.1 and file redirection (i.e. using the < and > symbols) is discussed in section 2.4 of A-8.

The *ja* utility (See section 6.3.6 also) may be used to time the example given above in the following manner:

```
ja
cft77 prog.f
segldr -o prog prog.o
prog < prog.in > & prog.out
ja -csft
```

The last use of *ja* is important, because it terminates the "job accounting" function of *ja* and gathers the statistics.

CRAY also supports a flowtrace mechanism for timing individual routines and a profiling utility called *prof.* They are both described in Documents CR-16 and CR-22. The flowtrace utility is described in the next section. There are two additional utilities available on the CRAY Y-MP only for monitoring program performance. The utilities *hpm* and perftrace rely on a hardware feature of **Sabre.** They are described in the sections immediately following the discussion of flowtrace. A FORTRAN-callable SECOND function is supplied by the FORTRAN library.

6.1.1. Flowtrace [c]

Flowtrace gathers information about subroutines during execution of the program; therefore, CPU overhead is generated. The type of output generated by flowtrace is determined by the option chosen, but may include:

1. Percentage of execution time spent in each routine.
2. Name or entry point.
3. Number of times a routine is called.
4. Calling routine (parent).
5. Time spent in routine exclusive of children.

To use flowtrace, the appropriate compiler option must be chosen and the appropriate library must be loaded. For the FORTRAN *cf77* load and compile command and the C compiler, cc, the option is *-F*. The *-F* option specifies flowtrace processing, including loading the library */lib/libflow.a.* If the *cft77* compiler and *segldr* are used separately, then the *cft77* command must use the *-ef* option and the *segldr* command must use the *-l flow* option.

After execution of the program, the command *flowview* must be executed to generate the statistical output. The command *flowview* can be executed under the X Window System (X). When using the command line options, you **must** specify the -L option to prevent the utilization of X and at least one other option. The option *-u* gives timings for all routines sorted in descending order by the amount of CPU time used by the routine. On **Voyager** and **Sabre** by default, flowtrace writes its data to a file called *flow.data.* The name of this file can be changed by setting the environment variable *FLOWDATA* to the name of the alternate file. The command *flowview* interprets the *flow.data* file and produces output statistics about the program's execution. On **Voyager** the *flow* command may still be used to interpret the *flow.data* file; however, the *flow* command will not be supported at UNICOS 7.0.

The following example generates a report in batch or interactive without using X:

```
cf77 -F prog.f
a.out > prog.out
flowview -Lu < flow.data > flow.out
```

It is not necessary to specify the file containing the flowtrace output unless it is called something other than *flow.data.*

The use of the FORTRAN subroutine FLOWMARK allows the selective timings of portions of code. The source code must be recompiled with two calls to FLOWMARK. The two calls to FLOWMARK mark the area of the code to be treated as a subroutine for statistical gathering. The argument of the first call is a seven character string. This is the name used in the flowtrace output to identify the timings from that part of the code. The argument to the second call is zero.

An example of a FORTRAN source code follows:

```
      CALL FLOWMARK ('partb'L)
      DO 10, i=1,N
      ...
10    CONTINUE
      CALL FLOWMARK (0)
```

The flowtrace output includes an entry *partb* which looks like the report for a subroutine.

Chapter 4 of *CRAY UNICOS Performance Utilities Reference Manual* (Document CR-16) gives more details, including the use of *flowview* under X and FLOWMARK.

6.1.2. Hardware Performance Monitor (hpm) [c]

The CRAY Y-MP hardware performance monitor, *hpm,* reports on the machine performance, during execution, of a program written in any language available under UNICOS. Unlike perftrace (See section 6.1.3), *hpm* reports only on whole programs, however it does not require a separate compilation with the flowtrace option. You may generate four types of reports: (0) execution summary; (1) hold-issue conditions; (2) memory use; or (3) vectorization and instruction summary. Generally reports (0) and (3) are the most informative. More information may be found in chapter 6 of *CRAY UNICOS Performance Utilities Reference Manual* (Document CR-16).

To use *hpm,* you compile and load your program normally. The *-g* option specifies which report you want. To get all four reports, you must execute your program under the control of *hpm* on four separate occasions. The output from *hpm* is sent to *stderr* and the default report is 0, execution summary.

The following example redirects the *hpm* output and the program output to file *hpm0.out:*

```
hpm -g 0 a.out >& hpm0.out
```

Because *hpm* output is sent to *stderr,* additional shell notation is required to separate *hpm* output from the program's *stdout.* The following is an example in the C shell. The *hpm* output is redirected to the file *hpm.out* and the program's output *stdout* is sent to *prog.out.*

```
(hpm -g 0 a.out > prog.out) >& hpm.out
```

6.1.3. Perftrace [c]

The perftrace utility gives the same type of statistics as *hpm,* but the results are broken down by each individual routine. It does not work with multitasked programs. To use perftrace, your program must be compiled with the flowtrace option (See section 6.1.1) and the library *perf* must be loaded. Since it uses flowtrace, there is considerable system overhead. After execution of the program, the command *perfview* must be executed to generate the report. The *perfview* command can be executed under X. When using the command line options, you **must** specify the *-L* option to prevent the utilization of X and at least one other option. The options *-Lu* give useful statistics for each routine sorted in descending order by the amount of CPU time used by the routines; however, many other command line options are available. By default, only report (0), execution summary, is generated.

The following example generates a typical useful report in batch or interactive without using X:

```
cf77 -F -l perf prog.f
a.out > prog.out
perfview -Lu > perf.report
```

However, perftrace can generate the other three reports by specifying the environment variable *PERF_GROUP* within the *env* command. To generate the vectorization report, you can use commands similar to the following:

```
cf77 -F -l perf prog.f
env PERF_GROUP=3 a.out > prog.out3
perfview -Lu > perf.report
```

The perftrace utility may be used with both C and PASCAL programs too. Chapter 7 of *CRAY UNICOS Performance Utilities Reference Manual* (Document CR-16) gives more detail, including the selective use of perftrace, other *env* environment variables and the use of *perfview* under X.

6.2. Program Optimization [c]

Optimization of programs on a multi-processing vector computer such as **Voyager** or **Sabre,** consists of scalar optimization, automatic vectorization and multitasking. Some optimizations are done automatically, some require compiler directives and some require manual code restructuring. The FORGE utility (See section 6.5) can be used to assist you in the restructuring of your code. These topics are discussed in the context of FORTRAN programming in the next three sections. Additionally, CR-37, *CRAY CF77 Compiling System, Volume 3: Vectorization Guide,* describes vectorization techniques in detail.

The key to optimizing a FORTRAN program is to identify the computationally heavy portions of the code. On CRAY supercomputers, one easy way of performing this task is to use the flowtrace utility (See section 6.1.1). Once the subroutines that consume most of the CPU cycles are identified, optimize them one or two at a time to reduce the chance of introducing an error. Any errors introduced by the attempted optimization are then easier to find and correct. This is especially important for multitasking (See section 6.3), since errors in task and data synchronization may produce results that are non-repeatable from run to run.

6.2.1. Scalar Optimization

The CRAY FORTRAN compilers, especially *cft77,* do an excellent job of scalar optimization. They recognize invariant code within a loop and remove it and calculate common subexpressions only once. Manual scalar optimizations, such as loop interchanging, loop unrolling, subroutine (or function) in-lining and use of PARAMETER statements for array dimensioning and DO loop limits improve a FORTRAN program's performance on a serial computer, as well as making automatic vectorization an easier task.

The *cft77* compiler eliminates both dead and useless code. If the results of a calculation are never used in another calculation, written to a file or stored into a variable in a COMMON block, then the code is deemed useless and eliminated. As a caveat to those who would test execution speeds on kernels lifted from other programs, this feature has been known to produce extremely high execution rates, when the kernel's code is deemed useless. The compiler may also reorder expressions for more efficient calculations. This can result in numerical differences. Use of the compiler directive $CDIR SUPRESS, before and after a statement can show whether a numerical difference occurred. Parentheses may be used to force expression evaluation in a particular order.

6.2.2. Automatic Vectorization [c]

Automatic vectorization is the default setting for the *cft77* compiler. It may be turned off on a loop-by-loop basis by using compiler directives. Only inner loops are candidates for vectorization; however, not all inner loops vectorize. In general loops with the following properties do not vectorize:

1. Any I/O statements.
2. CALL, PAUSE, STOP and RETURN statements.
3. References to CHARACTER data.
4. Backward branches within the loop.
5. Branches into the loop.
6. Three branch IF's or assigned and computed GOTO's.
7. Any recurrence, except vector reductions.
8. Ambiguous subscript references

Some of these problems may be resolved by restructuring the code. For example, if a loop contains a subroutine CALL, then often the loop can be split and the CALL moved to a separate loop. In some cases, the loop may be brought into the subroutine. Alternatively, short subroutines can often be expanded in-line within the DO loop.

The listing file contains informative messages about each loop that vectorized after every subroutine. Additionally each loop that didn't vectorize has a reason given for the failure. If a loop fails to vectorize, you may be able to resolve the problem by restructuring the code or by using a compiler directive.

If a loop has vectorized to the degradation of performance, scalar execution may be forced with the CDIR$ NOVECTOR compiler directive, which remains in effect until the end of the current program unit. If loops that appear later in the routine should still be vectorized, the CDIR$ VECTOR compiler directive should be used to turn vectorization on again. Some other compiler directives for optimization include:

CDIR$ IVDEP	Force the compiler to ignore potential vector dependencies in trying to vectorize the loop.
CDIR$ SHORTLOOP	Specifies that loop length is 64 or less.
CDIR$ NO SIDE EFFECTS	Specifies that called subroutine does not redefine variables local to the calling routine.
CDIR$ VFUNCTION	Specifies name of a vector version (written in CAL, the CRAY Assembly Language) of the scalar function referenced in the loop.

Chapter 1 of CR-35, *CRAY CF77 Compiling System, Volume 1: FORTRAN Reference Manual,* discusses the various compiler directives.

6.3. Multitasking [c]

LaRC's **Voyager** and **Sabre** are equipped with multiple identical and independent central processing units. **Voyager** has four central processing units and **Sabre** has five. Multitasking allows two or more parts of a program to be executed concurrently on these processors. The three forms of multitasking are macrotasking, microtasking, and autotasking, each of which is discussed in this section. In addition, brief discussions on code conversion, memory usage, and performance measurement are included. Detailed information on macrotasking and microtasking can be found in Document CR-18, *CRAY-2 Multitasking Programmer's Manual* and Document CR-33, *CRAY Y-MP, CRAY X-MP EA, and CRAY X-MP Multitasking Programmer's Manual.* Information on autotasking is located in Document CR-38, *CF77 Compiling System, Volume 4: Parallel Processing Guide.* Information on the UNICOS 6.1 X-based autotasking tools can be found in Document CR-16, *UNICOS Performance Utilities Reference Manual.*

6.3.1. Macrotasking [c]

Macrotasking was designed for programs with long execution times and large memory requirements running in a dedicated environment. Performance is best when macrotasking is used by programs containing large sections of disjoint code (or large-grain parallelism) which do not require a lot of synchronization. Macrotasking a program involves calling subroutines which exist in the macrotasking library, which is automatically loaded. The program modification for macrotasking results in code which is not portable across any machines other than CRAY computers.

Macrotasking is the more difficult form of multitasking to use, and some macrotasked programs can perform poorly on a heavily loaded system. For these reasons, macrotasking is not recommended to users.

6.3.2. Microtasking [b]

Microtasking is preferred over macrotasking for several reasons. The ease of code conversion makes microtasking an attractive choice. Microtasking a program requires inserting compiler directives, rather than subroutine calls, and as a result, microtasked programs maintain portability to other machines. Since the overhead for microtasking is smaller, tasks which are small and tightly coupled are efficiently multitasked. Parallelizing small tasks, usually outermost DO loops, makes data analysis much easier and less time consuming. Another advantage of microtasking is its design for batch environments. Microtasked programs can make efficient use of processors that are available for short periods of time. If no additional processors are available, little overhead is incurred.

The command to microtask a program is:

```
cf77 -Z m file.f
```

The *-Z m* option calls the microtasking preprocessor *premult* and specifies the use of microtasking libraries to the loader. The command above is equivalent to the following commands:

```
premult -F file.f
cft77 -a stack multf.f
as multc.s
cf77 multf.o multc.o -Z m
```

The executable in the example above is *a.out.* The *premult* option *-F* specifies that *cft77* is to be the compiler.

If *premult* is invoked explicitly, it leaves two files, *multf.f* and *multc.s* in the directory where it was executed. However, if *premult* is invoked by *cf77,* as in the first case above, then the files, *multc.s* and *multf.f* are automatically removed. The file *multc.s* contains CAL master routines for each microtasked subroutine. The file *multf.f* contains two subroutines for each microtasked subroutine. One subroutine is a multitasked version and the other is a single processor version. When a microtasked routine is called at run time, the CAL routine checks to see if microtasking is already in effect. If microtasking is in effect, further microtasking may not be invoked, so the single processor version of the routine is executed. These two versions of the original subroutine have as much of *mult* and *sngl* appended to their names as possible without making the new names more than eight characters long. Therefore, the original microtasked subroutine name should contain no more than seven characters. A microtasked subroutine name can contain eight characters as long as the *premult* option *-l* is selected. In this case, *premult* replaces the eighth character of the name with an *s* for the single processor version and an *m* for the multitasked version.

The file names *multf.f* and *multc.s* can be changed with the *-m* and *-c* options by entering:

```
premult -m filef.f -c filec.s -F file.f
```

The object files are created and loaded by entering:

```
cft77 -a stack filef.f
as filec.s
cf77 filef.o filec.o -Z m
```

Again, the executable is *a.out.*

Microtasked programs may also be created by entering:

```
cf77 -Z p file.f
```

If the program is partially microtasked, *fpp* analyzes any subroutines that do not contain microtasking directives. The translator, *fmp,* replaces both autotasking and microtasking directives with the appropriate code. The command above is equivalent to:

```
fpp file.f > file.m
fmp file.m > file.j
cft77 -b file.o -a stack file.j
as multc.s
cf77 -Z p file.o multc.o
/bin/rm file.m file.j file.o multc.s multc.o
```

Note that the assembler, *as,* must be invoked for the microtasked portion of the code.

As in macrotasking, the default number of CPU's allowed to execute a microtasked program is the number of system processors. The environment variable *NCPUS* is used to control the maximum number of CPU's which can execute microtasked programs. For example to request two processors, enter the following before executing the program:

```
setenv NCPUS 2
```

6.3.3. Autotasking [b]

Autotasking is the most recent form of multitasking. The autotasking compiling system provides the capability for automatic data analysis and compiler directive insertion. Autotasking may optimize some codes well, but autotasking cannot detect all forms of parallelism. User analysis and insertion of directives can lead to a significant increase in performance. As with all forms of multitasking, autotasking should be carefully applied to avoid adding unnecessary overhead.

Autotasking maintains the basic design of microtasking and includes some major improvements. Along with automatic data analysis and directive insertion, a major difference between autotasking and microtasking is the placement of parallel regions. In microtasked subroutines, the parallel region extends to the subroutine boundaries, and neither the main program nor functions can be microtasked. On the other hand, autotasking allows multiple parallel regions to be defined anywhere in the program. For example, an autotasked subroutine may have several sets of nested DO loops where each nested loop is defined as a separate parallel region. Any code outside of the parallel regions is executed by only one CPU. However, since the initiation of parallel regions requires a certain amount of overhead, the number of parallel regions should be limited.

The autotasking compiling system consists of three phases: dependency analysis, translation, and code generation. The dependency analysis phase, *fpp,* produces FORTRAN code optimized for vectorization and multitasking. Generally, innermost loops are analyed for vectorization and outermost loops are analyzed for concurrency. The translation phase, *fmp,* transforms the *fpp* output, replacing autotasking directives with the appropriate multitasking code. The code generation phase, *cft77,* produces machine executable code from the *fmp* output.

The command to autotask a program is:

```
cf77 -Z p file.f
```

The *-Z p* option specifies the use of *fpp,* the dependency analyzer, and *fmp,* the translator, before compilation with the *cft77* compiler. The executable resides in file *a.out.* The command above is equivalent to the following commands:

```
fpp file.f > file.m
fmp file.m > file.j
cft77 -b file.o -a stack file.j
cf77 -Z p file.o
/bin/rm file.m file.j file.o
```

The executable in the example above is *a.out*. The intermediate files, *file.m* and *file.j* may be retained by using the *-Z P* option. Refer to the *fpp*, *fmp*, and *cf77* man pages for additional command options.

The translation phase output file, *file.j*, can be significantly larger than the original FORTRAN file. This file contains master and slave code for each autotasked region. In autotasking and microtasking, a master process executes all code inside and outside of parallel regions; whereas, the slave processes execute code only within parallel regions. At run-time, the CPU executing the master process code checks to see if multitasking is being done at a higher level. If so, the master process executes a sequential version of the code in the parallel region. If multitasking is not being done at a higher level, the master process executes a multitasked version, and sends a signal which causes any additional connected CPU's to execute the slave process code.

There are a number of options available to the programmer which aid in improving the performance of *fpp*. For example, since *fpp* cannot analyze data across subroutine boundaries, loops containing subroutine calls are not autotasked. Inline expansion, controlled either through the command line or compiler directives, may increase the number of loops autotasked by *fpp*. Additional optimization techniques are described in the autotasking documentation.

6.3.4. Code Conversion [c]

Multitasked programs must execute using *stack* memory allocation mode, which allows the multiple CPU's to have separate storage locations for local variables. Under *stack* mode, local variables do not exist across subroutine calls, unless the FORTRAN SAVE statement is used. Since *static* is the default memory allocation scheme for **Voyager** and **Sabre,** verify that the program executes correctly in a *stack* environment before attempting multitasking. (On **Navier** and **Reynolds,** *stack* is the default mode.) The memory allocation may be changed from *static* to *stack* by the *cft77 -a stack* option. As mentioned earlier, if the *-Z m* or *-Z p* option is used, the program executes in *stack* mode.

Identify the time consuming routines of the program by using flowtrace (See section 6.1.1). Vectorize these routines as much as possible. Since greater performance improvements are obtained from vectorization, do not sacrifice vectorization for multitasking. In general, consider the outer DO loops for multitasking, since the inner loops may be vectorized.

If macrotasking, microtasking, or manually inserting autotasking directives, one must scope the data which involves determining if variables are shared or private. If the dependency analyzer, *fpp,* detects the parallelism, the data has been automatically scoped. Shared and private data must be used properly to obtain correct results. It may be necessary to use the *atscope* tool (See section 6.3.4.1) to analyze DO loops that *fpp* does not automatically parallelize for the scope of data within the loop. Shared data is known to all CPU's by one memory location, while separate copies of private variables exist for each CPU.

To verify correct execution, run the multitasked program on a single CPU, then multiple CPU's. Test the program in both batch and dedicated environments. To obtain dedicated runs, submit NQS scripts to the queue *multitask* by entering:

```
qsub -q multitask script_name
```

The multitasking queue is enabled a limited number of times per week. Refer to section 4.3.1 for details on NQS.

Multitasked programs are difficult to debug since errors are not usually reproducible. The programmer must make sure that the code is properly synchronized, since there is no certainty on the order in which parallel tasks are executed or which CPU's will actually execute specific parts of the code. Under UNICOS 6.1, improvements to the *cdbx* debugger (See chapter 5) allow users to set breakpoints, run to breakpoints, examine data, and perform other debugging functions on multitasked programs. The *atchop* utility (See section 6.4.3.2) may be helpful in determining the source of numerical differences between the autotasked and non-autotasked versions of a code.

6.3.4.1. atscope [c]

The X tool *atscope* assists in autotasking loops that *fpp* does not detect automatically to run in parallel. The tool displays text and provides a best guess as to the scope (shared, private, or unknown) of each variable in a loop. Clicking on a variable shows all occurrences of the variable in a loop. After all variables in a loop have been scoped, *atscope* inserts the appropriate autotasking directive.

To run *atscope,* enter:

```
atscope file.f
```

6.3.4.2. atchop [c]

The X tool *atchop* identifies subroutines and/or loops within subroutines that are causing numerical differences or abort conditions in programs which have been preprocessed by *fpp.* Before using *atchop,* you should determine if any reduction functions, such as inner products or summations, have been parallelized. The results obtained can be dependent on the order of execution if the data is of widely differing magnitudes. To check on the parallelization of reduction functions, use:

```
fpp file.f > file.m
```

The file, *file.m,* can be checked for autotasked reduction functions, which are always preceded by a *CMIC$ GUARD* directive. A sample *atchop,* session follows:

```
cf77 file.f
a.out < infile > out1
cf77 -Zp file.f
a.out < infile > out2
diff out1 out2 > outdifs
atchop -Zp -r infile -c out1 -b file.f
```

If the file *outdifs* has zero length, then no numerical differences were introduced by autotasking. The *atchop* command line options used above are:

Zp	invoke all phases of the compiling system
r	designates the user's standard input file
c	designates file holding sequential results
b	performs both a binary and fpp chop

By default, *atchop* compiles, loads and executes in a temporary directory in */tmp.* The location of this directory can be controlled with the *TMPDIR* environment variable.

6.3.5. Memory Usage [c]

Under UNICOS 6.1, the individual task stack size is computed which should lead to reduced memory requirements for autotasked programs. The following example describes how the stack memory requirements for autotasked programs may be lowered if necessary.

The over estimation of autotasking memory requirements can occur when when large private arrays are used. A possible solution to this problem is to place large local arrays in FORTRAN COMMON or SAVE statements. Another alternative is to redefine the initial and incremental stack sizes. A *segldr* load map can be generated with the following statement:

```
cf77 -Zp -Wl,-D'MAP=STAT' file.f
```

The *cf77* option *-Wc,arg1[,arg2][,...]* passes arguments to various phases of the compiling system. In the command above, *l* corresponds to the loader. (Options may also be passed to *premult, fpp* and *fmp* by using *-Wm, -Wd* and *-Wu* respectively.)

In the load map, locate the initial decimal stack size under the memory statistics section. Next, analyze the subprogram units and estimate the largest amount of stack space needed by one processor for private variables. For example, assume that the load map gives an initial stack size of 2000000 and the stack size estimate for the program is 10000. The memory management can be controlled by the following command:

```
cf77 -Zp -Wl,-D'STACK=10000+1990000' file.f
```

As a result, each task initially receives 10000 words, rather than 2000000. When the master task needs more memory, it receives 1990000 (2000000 - 10000) additional words of memory. Care should be taken to avoid underestimating the initial stack size.

The stack space is contained within the the dynamic memory area, also known as the heap. Since the initial heap size is based on the initial stack size, the initial heap size should also be reset. Otherwise, many requests must be made in order to allocate memory for the master task. For example, if four processors are used, the initial heap can be estimated with the following:

*initial_heap = (10000 * 4) + 1990000*

Both the stack and heap values can be changed with the following command:

```
cf77 -Zp -Wl,-D'STACK=10000+1990000;HEAP=2030000' file.f
```

6.3.6. Performance Measurement [c]

The goal of multitasking is to divide a program's CPU time among multiple processors, thus reducing the actual elapsed time. Speedups are calculated by comparing the wall clock time of the multitasked program with that of the sequential program. Before attempting multitasking, determine the amount of parallelizable code in the program. Given the percentage of parallel code in a program, Amdahl's Law (See CR-18 or *man amlaw* for more information) predicts the theoretical maximum speedup in a dedicated environment. In addition to sequential code, several factors including the overhead required for the multitasking code and any load imbalance across parallel tasks contribute to speedup degradation. Multitasked programs containing well balanced tasks with large grain parallelism generate less overhead.

The performance of a multitasked program running in a batch environment is difficult to measure and varies from one run to the next depending on the system load. However, even on heavily loaded systems, multitasked programs usually receive some benefit. In general, this is true for microtasked and autotasked programs; however, on some CRAY systems, macrotasked programs can behave poorly. A macrotasked program containing many synchronization points requires extensive task management and may result in an elapsed time much greater than the time required for the sequential program. See section 6.3.6.1 for a discussion of the *atexpert* graphical tool for displaying performance information about an autotasked program.

The elapsed time and CPU time for an entire program may be displayed with */bin/time* (See section 6.1). Additional timing information may be displayed with the job accounting utility, *ja.* To produce a report for the executable file *program,* enter the following commands:

```
ja
program
ja -st
```

The option *-t* terminates job accounting, while the option *-s* produces the job accounting summary report. Included in this report is the time spent executing on *n* processors concurrently. This CPU timing breakdown can also be displayed by calling the subroutine *MTTIMES* at the end of program execution. Appendix A of CR-18 gives an example of the output from *MTTIMES.* This subroutine returns the amount of time that the program spent executing with 1, 2, 3 or 4 CPU's, as well as total CPU time and a measure of how much overlap there was during program execution. All timings of multitasked code are dependent on system load and may vary from run to run.

Other functions exist for timing a FORTRAN program. *SECOND* returns the cumulative CPU time in seconds, while *IRTC* and *TIMEF* measure wall clock time in clock periods and milliseconds respectively. On **Voyager,** one clock period is 4.1E-9 seconds (4.1 nanoseconds), and on **Sabre,** the clock period is 6.17 nanoseconds. For example, to time SUBROUTINE A:

```
W1= TIMEF()
T1= SECOND()
CALL A
T2= SECOND()
W2= TIMEF()
CPU= T2 -T1
WALL= (W2 -W1)/1000.
```

If SUBROUTINE A is microtasked and multiple CPU's are available then the time returned in *WALL* should be less than the time returned in *CPU.*

6.3.6.1. atexpert [c]

The *atexpert* graphical tool displays performance information of an autotasked program based on statistics gathered during execution of the program on an arbitrarily loaded system. The display shows how the program actually performed by showing the serial and parallel times for each parallel region. Serial time outside of parallel regions and a breakdown of autotasking overhead is also provided. Two speedup curves are shown on the *atexpert* display. The top curve displays the speedup calculated from Amdahl's Law and the bottom curve displays the predicted dedicated speedups for a given number of CPU's. The Amdahl's Law curve shows the speedup attainable with the percentage of parallelism exploited in the program assuming there is no autotasking overhead. The dedicated speedups are calculated using measured values of parallel time and sequential time. Large gaps between the ideal speedup (the number of CPU's used) and the Amdahl's Law curve indicate a large amount of serial code present in the program. A large gap between the Amdahl curve and the dedicated curve indicates that overhead is affecting parallel performance.

To use *atexpert,* enter the following commands:

```
cff77 -Zp -Wu"-p" file.f
atexpert -f atx.raw
```

The *-Wu"-p"* option causes *fmp* to generate output for use with *atexpert.* The *-f* option specifies the file with which *atexpert* is to work. Reports detailing the speedups and overheads for the program, subroutine, and loop levels can be generated using the *-r* option.

6.4. Source Code Control System

The Source Code Control System (SCCS) is a collection of utilities running under UNICOS. SCCS tracks modifications to files. The following capabilities are provided:

1. Storing files of text.
2. Retrieving particular versions of files.
3. Controlling updating privileges to files.
4. Identifying the version of retrieved files.
5. Recording when, where and why a change was made.
6. Identifying the author of a change.

SCCS works on source code or text files, but not on binary files or executables. It uses a control file to accomplish all of the above tasks. Document CR-12, *CRAY UNICOS Source Code Control System User's Guide,* describes SCCS in detail.

The control file's name begins with the characters *s.,* and is created with the *admin* command, as follows:

```
admin -isource.f s.source.f
```

The file *source.f* initializes the original control file. The *get* command retrieves the latest version of the source code (or text file) from the control file, as shown:

```
get s.source.f
```

The latest version of *source.f* is recreated, and the version number of the file and number of lines of text are output to the screen. The *delta* command is used to record the changes made to a file during an editing session. The comments added are limited to 512 characters. Enter

```
delta s.source.f
```

and you are be prompted with

```
comments?
```

To continue a line of comments, end the line with a back slash (\) and a carriage return.

Other SCCS commands are described in chapter 5 of CR-12, while chapter 6 discusses file formats, file maintenance and access permissions.

6.5. FORGE [d]

FORGE is an interactive program global analysis system, that was developed by Pacific-Sierra Research (PSR) Corporation and is now supported by Applied Parallel Research (APR). It has an user interface to the X Window System (X). FORGE provides tools that allow you to analyze a program and to use that information to transform the program into a more efficient code. It has an instrumentation facility that allows you to time selected subroutines down to the DO loop level. It is well suited as both a utility to improve the performance of "dusty deck" codes and an environment within which new efficient code can be developed. *The FORGE User's Guide,* Document CR-32, is the basic guide to the baseline FORGE system.

FORGE builds and maintains a database of all variable usage and flow of control, which is compiled from your original program. With this database, you can trace the use of variables across any part of the program calling tree, including implicit and explicit equivalencing. It also contains features that allow code reformatting and version control for experimental versions of subroutines. The X interface is menu driven and has a HELP utility. Section 3.4 of the *SNS Programming Environment User's Guide* describes the steps that need to be taken for you to access a utility on a remote machine with an X interface.

FORGE is available for all Sun Microsystem SPARC workstations at LaRC. The FORGE SPARC-executable code is in the directory

```
~tennille/FORGE/forge.tar
```

on **Eagle.** Installation of FORGE on your Sun workstation may require the assistance of your System Administrator if you do not have write permission in the */usr/local* directory. The recommended installation point is in the */usr/local/FORGE* directory, which may be accomplished with the following commands:

```
cd /usr/local
mkdir FORGE
cd FORGE
tar -xvf forge.tar
```

The following alias is recommended for your *.cshrc* file

```
alias forge "/usr/local/FORGE/xforge -f /usr/local/FORGE"
```

to insure that FORGE can locate all necessary HELP files.

6.5.1. Using FORGE with the X Interface [c]

When you invoke FORGE, there are several "How To Use ..." entries that appear in the right window called Main Menu. The most efficient manner to access these entries is to use your mouse to "pick" them by clicking the left button. To use FORGE to analyze a code, you must create a FORGE "package", which is simply a set of files and directories maintained in a directory named *psr.dir,* which is created for you automatically. These files may become rather large.

You must identify which files are to be included in a "package". FORGE performs a cursory parsing of the identified files, which includes locating the beginning and end of each subroutine. Once the "package" is created, you may invoke the FORGE code reformatting utility (See section 2.5.6 of CR-32) or instrumentation facility (See section 2.5.5 of CR-32). When you time routines on **Voyager** and **Sabre** with the instrumentation facility, the library *~tennille/FORGE/psrtim.a* must be loaded with your code to gather the statistics.

Since FORGE is menu-driven, it is relatively easy to learn your way around the system. In general, the left mouse button is used to "pick", the right for "show", and the middle for "help" on the item selected. The Main Menu has the following selectable items:

Package Creation and Selection
Analyze and Modify Current Package
Change Directory
How To Use FORGE
How To Use the Mouse
How To Use Command Mode
Options
Exit

This menu appears whenever you invoke FORGE or "pick" the MENU box in the upper right corner of the display. When you "pick" any of the above items, a new menu pops up for you to chose another item. By default, the left window displays the menu "Analyze and Modify Current Package" when FORGE is invoked.

Within the "Analyze and Modify Current Package" menu is an entry titled "Define/Edit Files in Package". When you pick this item, one of the entries in its menu is "Select Hardware File". "Pick" this item, and you may choose the target hardware for which you want to optimize your code, so even though FORGE is only available on **Voyager,** you may also optimize codes for **Sabre.**

The power of FORGE lies in its ability to do a global analysis. You may query the database built by FORGE using templates as illustrated in section 2.5.3.4 of CR-32. A template is simply a filter to specify the usage of variable. For example, you might wish to locate all variables that are used somewhere but never defined or conversely variables that are defined but never used.

Appendix A of CR-32 describes the setting of various options to control your FORGE environment. Appendix B describes the command line options. Appendix D is a sample X interface FORGE session.

7. CRAY DOCUMENTATION [d]

Users of the CRAY supercomputers have several sources of information and assistance (see chapter 8 of A-8). UNICOS 6.1.6 provides much more information on-line than UNICOS 5.1. However, ACD still provides substantial printed documentation that is described below.

ACD automatically distributes several CRAY manuals (highlighted in **boldface** in **Table 7.1** on the next page) that describe frequently used features of the CRAY supercomputers to **SNS** Document Librarians. The only manual that is new or revised is CR-61, which reflects the hardware upgrade to **Sabre.** All manuals in the table are available on an individual basis from OCO by calling 864-6562; visiting room 1035 in Building 1268; or by sending electronic mail to *oco@eagle,* for overnight service. This list is current as of January, 1993. Refer to *notes tradoc* on **Eagle** for any later documentation changes. Two of the manuals have been compiled by ACD personnel: CR-1, *CRAY Mini Manual;* and CR-3, *CRAY Mathematical Libraries.* Each newly validated **SNS** user receives copies of CR-1, CX-1, the *CONVEX Mini Manual* and A-8, the *SNS Programming Environment User's Guide.* There are sufficient quantities of CR-2 and CR-35 in OCO for any user to obtain a personal copy. Demand determines the number of copies of other manuals that are kept in stock by OCO for individual distribution. The CONVEX series manuals CX-19 and CX-22, which describe the *notesfile* and *make* utilities, are also available from OCO.

7.1. The CRAY UNICOS Primer [c]

Document CR-2, the *CRAY UNICOS Primer,* is a useful manual for the novice UNICOS programmer and has been rewritten for UNICOS 6.1. It is designed to assist you with the following:

1. logging to a CRAY system.
2. using basic UNICOS commands.
3. communicating with other users.
4. creating files.
5. compiling programs.
6. understanding shell scripts and environment variables.
7. describing the hierarchical file system.

CR-2 may be used as a self-paced tutorial on CRAY UNICOS.

CSCC Doc No	Title	CRAY Doc No
CR-1c	**CRAY Mini Manual (April 1992)**	
CR-2a	**CRAY UNICOS Primer**	**SG-2010 6.0**
CR-3a	**CRAY Mathematical Libraries (January 1990)**	
CR-5a	CRAY FORTRAN (CFT2) Reference Manual	SR-2007D
CR-6b(v1)	**CRAY UNICOS User Commands Reference Manual, Volume 1**	**SR-2011 6.0**
CR-6b(v2)	**CRAY UNICOS User Commands Reference Manual, Volume 2**	**SR-2011 6.0**
CR-7	**CRAY UNICOS Editor's Primer**	**SG-2050**
CR-8b	CRAY Segment Loader (SEGLDR) and ld Reference Manual	SR-0066 6.0
CR-9a	CRAY UNICOS Support Tools Guide	SG-2016 6.0
CR-10	CRAY Symbolic Debugging Package Reference Manual	SR-0112C
CR-11b	CRAY TCP/IP and OSI Network User's Guide	SG-2009 6.0
CR-12	CRAY UNICOS Source Code Control System User's Guide	SG-2017
CR-13a	CRAY-2 UNICOS Macros and Opdefs Reference Manual	SR-2082 6.0
CR-16b	**CRAY UNICOS Performance Utilities Reference Manual**	**SR-2040 6.0**
CR-18a	CRAY Multitasking Programmer's Reference Manual	SN-2026C
CR-20	CRAY C Reference Manual	SR-2024
CR-21a	CRAY PASCAL Reference Manual	SR-0060 4.2
CR-22	CRAY Computer Systems User Environment	SN-2086
CR-23a	CRAY UPDATE Reference Manual	SR-0013K
CR-24	CRAY SORT Reference Manual	SR-0074
CR-25a	**CRAY UNICOS CDBX Debugger User's Guide**	**SG-2094 6.0**
CR-26b	CRAY UNICOS CDBX Symbolic Debugger Reference Manual	SR-2091 6.1
CR-27a	CRAY UNICOS FORTRAN Library Reference Manual	SR-2079 6.0
CR-29a	**CRAY UNICOS Math and Scientific Library Reference Manual**	**SR-2081 6.0**
CR-30a	CRAY Standard C Programmer's Reference Manual	SR-2074 3.0
CR-31a	CRAY C Library Reference Manual	SR-2080 6.0
CR-32b	CRAY The FORGE User's Guide	
CR-33	CRAY Y-MP, CRAY X-MP EA and CRAY X-MP Multitasking Programmers' Manual	SR-0222F-01
CR-34	CRAY Macros and Opdefs Reference Manual	SR-0012D
CR-35a	**CF77 Compiling System, Volume 1: FORTRAN Reference Manual**	**SG-3071 5.0**
CR-36a	**CF77 Compiling System, Volume 2: Compiler Message Manual**	**SG-3072 5.0**
CR-37	**CF77 Compiling System, Volume 3: Vectorization Guide**	**SG-3073 5.0**
CR-38a	**CF77 Compiling System, Volume 4: Parallel Processing Guide**	**SG-3074 5.0**
CR-39	CRAY UNICOS Source Manager (USM) User's Guide	SG-2097 6.0
CR-40	CRAY UNICOS X Window System Reference Manual	SR-2101 6.0
CR-41	CRAY DOCVIEW User's Guide	SG-2109 6.0
CR-42	CRAY DOCVIEW Writer's Guide	SG-2118 6.0
CR-43	CRAY UNICOS I/O Technical Note	SN-3075 6.0
CR-61a	CRAY Y-MP Functional Description Manual	HR-04016-0A
CR-76	CRAY-2 Computer System Functional Description	HR-2000C

Table 7.1 - CRAY Documentation [d]

CR-1 SECTION GUIDE TO HIDDEN FILES [b]

File	Description	Section
.cshrc	Executes when a C-shell is spawned	4.1.5
.login	Executes when you log into a UNIX machine	4.1.4
.logout	Executes when you log off a UNIX machine	4.2.3

CR-1 SECTION GUIDE TO COMMANDS [c]

Command	Description	Section
admin	create a SCCS control file	6.4
alias	create an alias	4.1.3
atchop	check for numerical differences from autotasking	6.3.4.2
atexpert	display performance statistics for autotasked jobs	6.3.6.1
atscope	scope loop variables for autotasking	6.3.4.1
cc	invoke Standard C compiler	3.2
cdbx	invoke interactive symbolic debugger	5.1
cft77	invoke FORTRAN-77 compiler	2.1
cf77	invoke cft77 compile & load	2.3
debug	invoke batch post-mortem debugger	5.3
debugx	invoke debug automatically	5.5
err.exe	utility to locate FORTRAN line number	5.4
exit	terminate an interactive session	4.2.3
explain	interpret error message	5.6.1
flowview	generate flowtrace output	6.1.1
hpm	invoke hardware performance monitor	6.1.2
ja	obtain job accounting information	6.1
kill	terminate a process	4.2.2
logout	terminate an interactive session	4.2.3
nohup	allow interactive process to run after logout	4.2.1
pascal	invoke PASCAL compiler	3.1
passwd	change your password	4.1.1
pcc	invoke portable C compiler	3.3
perfview	generate performance statistics report	6.1.3
premult	invoke microtasking preprocessor	6.3.2
printenv	check on status of environment variables	4.1.2
qdel	remove running or queued NQS job	4.3.1
qlimit	display NQS batch limits	4.3.1
qstat	display status of NQS jobs	4.3.1
qsub	submit a job to NQS	4.3.1
segldr	invoke the segment loader	2.2
set	set C-shell variable	4.1.2
setenv	set environment variable	4.1.2
time	check execution and wall time	6.1

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE January 1993	3. REPORT TYPE AND DATES COVERED Technical Memorandum

4. TITLE AND SUBTITLE CRAY Mini Manual	5. FUNDING NUMBERS 505-90-53-02
6. AUTHOR(S) Geoffrey M. Tennille and Lona M. Howser	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) NASA Langley Research Center Hampton, VA 23681-0001	8. PERFORMING ORGANIZATION REPORT NUMBER CSCC Doc. No. CR-1d
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA TM-107599 (revised)

11. SUPPLEMENTARY NOTES
Tennille and Howser: Langley Research Center, Hampton, Virginia.
This TM supersedes the April 1992 version (CR-1c)

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified - Unlimited Subject Category 60	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This document briefly describes the use of the CRAY supercomputers that are an integral part of the Supercomputing Network Subsystem of the Central Scientific Computing Complex at the Langley Research Center. Features of the CRAY supercomputers are covered, including: FORTRAN, C, PASCAL, architectures of the CRAY-2 and CRAY Y-MP, the CRAY UNICOS environment, batch job submittal, debugging, performance analysis, parallel processing, utilities unique to CRAY and documentation.

The document is intended for all CRAY users as a ready reference to frequently asked questions and to more detailed information contained with the vendor manuals. It is appropriate for both the novice and the experienced user.

This revision reflects hardware upgrades to the CRAY-Y-MP, changes to operational procedures and software.

14. SUBJECT TERMS FORTRAN, multitasking, debugging, computing environment	15. NUMBER OF PAGES 78
	16. PRICE CODE A05

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT