Centre for High Performance Computing & Visualization

Local Cray Guide

February 2002

Rob de Bruin & Doeko Homan

Local Cray Guide

TABLE OF CONTENTS

1. Introduction
- HPC&V
- Facilities
- How to Contact us

2. Site dependencies
- Introduction
- Security
- Accounting
- Quota
- Running Jobs
- System Status

3. Architecture
- Introduction
- Cache
- Memory
- Processors
- Hardware Performance Counters
- Multi-Streaming Processor Directives
- Network
- Back up
- Mass Storage
- Performance

4. Software
- Programming Environments
- Fortran
- Debuggers
- Performance Analysis Tools
- Libraries
- Documentation

5. Optimization
- Basic Consepts
- Cray Multi-tasking
- Parallelization Levels
- Parallel Processing Terminology
- Compiler Autotasking
- Parallel Processing Directives
- ATExpert

6. Support
- Training
- Consultancy
- Projects

Appendix A
- Bit Matrix Multiply

Appendix B
- Some F90 and F77 implementation differences

1. Introduction

1.1 Centre for High Performance Computing and Visualization

High Performance Computing (HPC) as well as Visualization and Virtual Reality (VR) are two areas of the RuG-ICT strategy that receive a new impulse in order to continue a tradition to provide Groningen scientists with state-of-the-art facilities needed for excellence in computational science.

The mission of the Centre for High Performance Computing and Visualization (HPC&V) is to support and stimulate the use of facilities for these two areas of the RuG-ICT strategy. These facilities should benefit all scientific communities within the University of Groningen.

HPC&V is one of the three national centres for HPC appointed by NOW (National Scientific Research). The High Performance Computing facilities are partly financed by NCF (National Computer Facilities). Industrial or governmental departments, not from RuG, can use the facilites too.

HPC&V is part of the Computing Centre of the University of Groningen (RuG). A Scientific Board consisting of scientists of the RuG advises the management of the HPC&V on technical and scientific issues.

1.2 Facilities

At present the HPC-facilities, consists of a 22 node Linux Alpha cluster having a peak performance of 22 Gflop/s (February 2000), a 132 node Linux Intel cluster of 250 Gflop/s (September 2001, will be upgraded to a 500 Gflop/s system in 2003), a 16 node SGI Onyx 3400 (July 2001) of 16 Gflop/s and a vector supercomputer Cray SV1e of 64 Gflop/s and 32 Gbytes shared memory (December 2001).

The new to be built Zernikeborg will host a high-end virtual reality system consisting of a CAVE and a Reality Theatre, driven by the 16 node four pipe SGI Onyx 3400. The building and facilities are in operation from July 2002.

A preliminary version of the VR facility with one screen (Powerwall) and the SGI Onyx 3400, is available since July 2001. Visualizations can be seen in stereo on the screen using shutter glasses. Future CAVE and Reality Theatre users can prepare their applications on this prelimary facility.

This paper is intended for (novice) users of the Cray SV1e.

1.3 How to contact us

Visiting address	Nettelbosje 1, 9747AE, Groningen, The Netherlands
P.O. Box	11044, 9700CA Groningen, The Netherlands
Mail	*HPCV@rc.rug.nl*
Telephone	+31 50 3638080
Fax	+31 50 3633406
Internet	www.rug.nl/hpc

2. Site Dependencies

2.1 Introduction
It is intended to give new users fast and easy access to the Cray Sve1. Three groups of users are distinguished: RuG personnel and students, scientific users administered by the NCF (the Dutch foundation for national computer facilities), and others.

- For RuG users it is sufficient to provide HPC&V some personalia and a valid, active email address.
- For NCF users HPC&V also needs these personalia and email address and proof of the awarded compute resources by NCF.
- Other potential users should contact HPC&V.

For the first two groups of users there are no financial consequences.

2.2 Security
On a regular basis a Security Scan is run. The Cray SV1e shows no vulnerabilities. HPC&V does *not* give any warranty and can never be hold responsible for any damage that is caused to the user from whatever source. At the moment there is no SSH available.

2.3 Accounting
After subscription, your account will be opened within one working day. The provided password needs to be changed within the next working day.
There are three important rules of the game:

- With an account the user's valid, active email address is attached. If the email address can not be reached, or the owner of the email address does not respond within a reasonable time span, HPC&V can remove the user's files.
- When an account has not been used for a substantial period of time, the user can be asked to remove large files. In case the user is not responding, HPC&V will remove the files. This rule will apply until an archiving system is installed.
- It is very much discouraged that the user passes on the account to someone else (in his group). Besides for obvious reasons, it happens every now and then that the 'old' users – which are kept informed on changes in the system - do not pass this information to the 'new' ones, causing serious problems to the 'new' user.

To login, a telnet session is started

telnet cray.service.rug.nl

2.2 Quota
User Culture
This section is about those aspects of friendly user behavior that cannot be dictated by the system the way some quota, allocation, and the number of jobs in the queue can be controlled. In general, we attempt to provide an atmosphere of freedom of use with the intent to accomplish required goals.
Resources as disk space, memory, CPUs, etc. are large but limited. Where more users are active on one machine, resources become even more limited. Such is life. Usually, on top of that, administration limits the user's quota even further. This is *not* the local policy: there are no limits to the access of disk-space, number of CPUs, memory, etc. on the Cray SV1e. We over-allocate certain aspects to allow maximum use by those who are currently active. Of course, in this limitless world we need the cooporation, fairness and discipline of the users.

This policy turns out to be succesful - and very efficient in terms of resources - for many years now at RuG. (HPC&V may alter this policy whenever they think it is needed.)
Using the **df –kP** command one gets an overview of the disk partitions. Most important are **/u0** and **/u1** (user home directories – 124 Gbytes) and **/tmp** (for temporary files – 9 Gbytes). On **/tmp** the files needed for execution are located (both interactive and batch jobs). For further information use **echo $TMPDIR.** After completion of the job, **$TMPDIR** is removed. There is no quotation on **/tmp**. Its smooth use is left to the users. So, maintain your disk space. The "superhome" file system provides ample space for storing your active files, however, on-line space is limited. Files on **/tmp** can be removed without warning and moreover no back-up is made of these files!

2.5 Running Jobs

Interactive jobs

Interactive jobs on the Cray SV1e are limited to half an hour CPU time per process (i.e. the time consumed by all processors involved), but there is no limit to the CPU-time used during your interactive session. The maximum memory space for interactive jobs is 64 Mwords (0.5 Gbyte). The main reason for this limitation is in the fact that the Cray has no virtual memory. Jobs in need for more time or memory must be run as batch-jobs (queued).

Batch jobs

For the execution of batch jobs, NQS (the Network Queueing System) is used. To submit a job, one has to place all necessary commands in a script-file, say **script,** then type the command **qsub script.** To this command additional parameters can be added that limit total CPU time and memory requested. Depending on these values the job is placed in a specific queue (class). For CPU time three classes are defined, for memory there are two classes.

CPU time is divided into

0-5 hours	**short** queue; default,
5-50 hours	**long** queue,
>50 hours	**verylong** queue; the maximum CPU time must be given by the user as parameter in the **qsub** command, e.g. **–lT 360000**, i.e. 100 CPU hours.

One should provide a fair estimate of the time your job needs, since this is also useful information for other users queueing after you.

At the moment memory space is devided into

Upto 128 Mwords	1 Gbytes; default,
Upto 512 Mwords	4 Gbytes) per job. The maximum memory space must be given by the user as parameter in the **qsub** command, e.g. **–lM 2Gb**, for 2 Gbytes). With **jobq** one can examine the actual memory space used.

Examples

Job needs less than 5 hours of CPU time and less than 128 Mwords memory:

qsub script

Job needs 100 hours of CPU time and 2Gb memory:

qsub –q verylong –lT 360000 –lM 2Gb script

For further information about queueing see: **man jobq** and **man qsub**.

A job can be deleted from a queue with **qdel.** An executing job can be aborted with **qdel –k,** see **man qdel**.

Remark 1: Users may want to use the Cray SV1e all to themselves, e.g. to study accurate timing of their codes. To this purpose a **single** job queue exists. This queue is activated only after personal communications with the machine's administrator!

Remark 2: For extensive a posteriori information about the job in terms of CPU time and memory usage (but also other information e.g. on parallellization) use **ja** (job accounting). For instance, you may add the following in the script file

```
ja
./a.out
ja -cst
```

(Also consider **hpm**: e.g. **hpm –g0 ./a.out**)

2.6 System Status

Maintenance of the machine is scheduled on Monday mornings if necessary. To restart batch jobs again, checkpoint restarting is used. This usually works and is transparant to the users. For some specific code checkpointing is not possible. It is therefor a good policy in case of very large jobs (e.g. 1000 CPU hours or more), to add to your code some kind of home-made checkpointing i.e. to break down the job in smaller chuncks and store intermediate results.

Experience with the former Cray learned that down time due to system failures is neglectable.

HPC&V generates its own power in case of regional power network failures. The Cray SV1e is also powered by this no-break installation.

3. Architecture

3.1 Introduction

Most aspects of using RuG's new Cray Inc. SV1e supercomputer, are just like RuG’s Cray Research J90. The compilers and libraries are a more recent version, and there are faster processors and more memory, but the UNICOS operating system and local additions including the batch scheduling system should work as they did on the J90.

This paper is intended to highlight what's different on the SV1e rather than tell you everything there is from scratch.

3.2 Cache

The Cray SV1e computer is significantly different from previous Cray vector machines in that it provides a cache for the data resulting from scalar, vector, and instruction buffer memory references. Traditional Cray vector computers as the J90 had no intermediate data cache between memory and the vector and scalar registers, so all operands were fetched directly from memory for all operations.

With the SV1, each CPU has a 32 Kword cache in which both scalar and vector stores and loads are cached. The latency for fetching data in this cache is four to five times less than that of fetches directly from memory. In addition, when a processor fetches data from its own cache instead of from the memory, this reduces contention with other processors for memory access.

This cache is four way set associative. This means that a given cache line can reside in one of four locations within the cache. A memory address is mapped to a given cache line based on the last 13 bits of its physical address, so every 8192 words of memory are mapped to the same four way cache line. You should avoid memory accesses with strides that are a multiple of 8192, since four of these will fill up the cache slot. In general, memory accesses whose strides are large powers of two are to be avoided, since e.g. only eight loads of stride 4096 and sixteen of stride 2048 will fit in cache. So, on the non-cached J90, performance on vector constructs generally increases as a predictable function of vector length. This is not always the case on the Cray SV1e system, since long vectors can lead to a reduction in data cache efficiency.

- cache size per processor - 256Kbytes = 32Kwords
- cache line width - 8 bytes = 1 word
- scalar references prefetch 8 words to cache
- 4 way set associativity
- write allocate, and write-through cache
- least recently used replacement (LRU)

3.3 Memory

RuG’s Cray SV1e is configured with 32Gbytes, i.e. 4Gwords of uniform access, shared central memory. This is 8 times the size of memory of the former J90. Of this 4Gwords of memory, each user application is limited (see Section 2.5 Running Jobs).

- uniform memory access, i.e. access time for any CPU memory reference to any location in memory is the same
- all physical memory, no virtual memory
- 4 Gwords of main memory
- number of memory banks = 1024

- system memory = 74,713,088 words
- user memory = 42,120,251,904 words
- memory is word-addressable

Use for futher details **sysconf.**

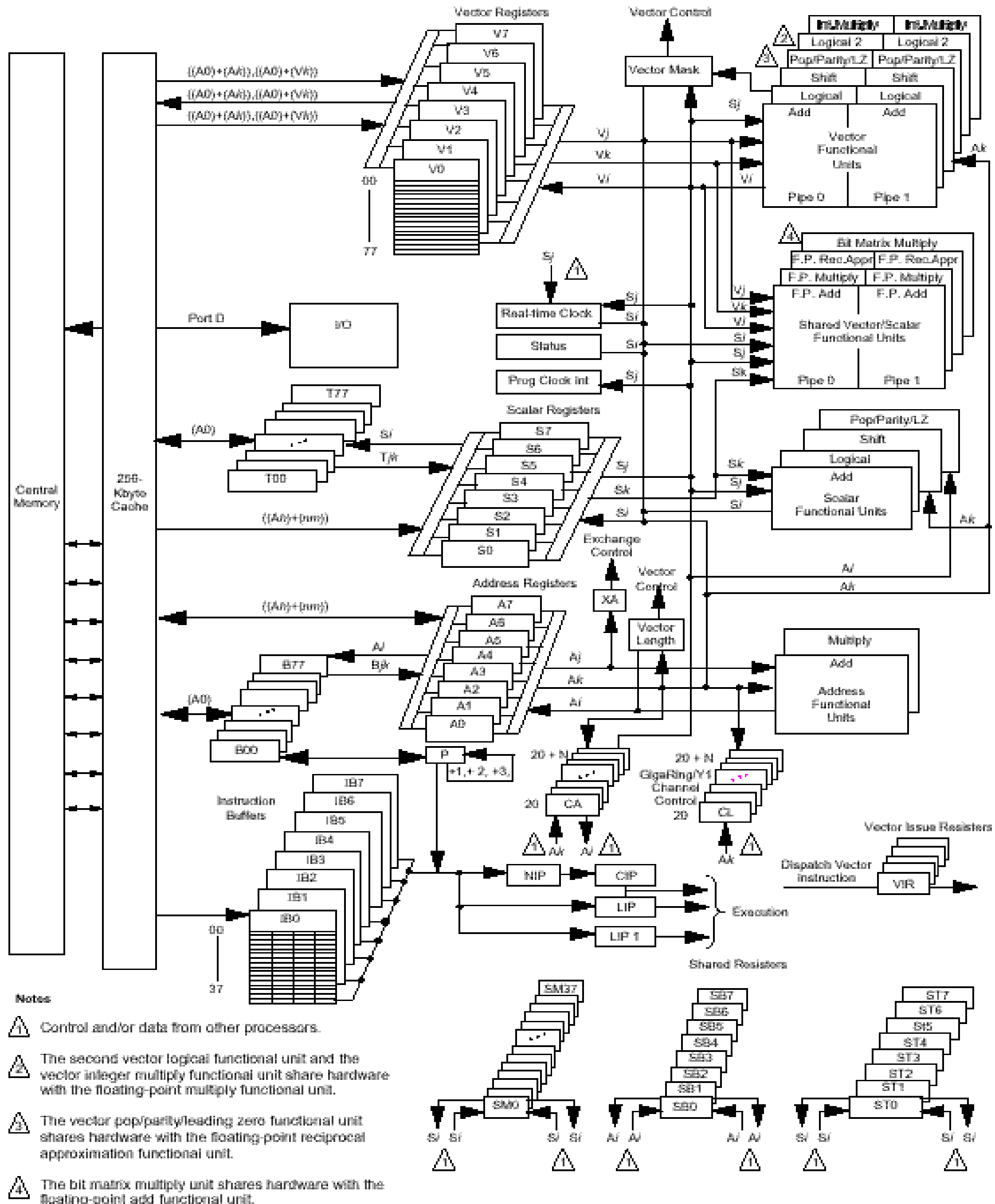
3.4 Processors

The Cray SV1e originates from the Y-MP and J90 PVP (Parallel Vector Porcessor) product lines. It has 32 central processing units (CPUs). The clock speed of each processor is 500 MHz. Each CPU has several dual-pipelined functional units that can deliver 4 floating-point results per CPU clock cycle. With a 500MHz CPU clock the peak floating point rate per CPU is 2.0 Gflop/s and 64 Gflop/s for the 32 CPU system.

In addition to the CPU clock there is a system clock which runs at the rate of 100 MHz.

Some of the hardware included in each processor is listed below. The block diagram for a single processor provides more details.

Registers
8 vector (V) registers, each holds 64 elements
8 scalar (S) registers
8 address (A) registers
64 T registers
64 B registers
8 shared B registers
8 shared T registers
Vector functional units
floating point add - 2 pipes
floating point multiply - 2 pipes
reciprocal - 2 pipes
bit matrix multiply - 1 pipe only
logical - 2 pipes
2nd logical - 2 pipes
shift - 2 pipes
pop/parity/leading zero - 2 pipes
integer add - 2 pipes
integer multiply - 2 pipes
Two memory requests/clock cycle
two dual-port reads (4 words)
one dual-port read and one dual-port write (4 words total)
one dual-port write only (2 words)



3.5 Hardware Performance Counters

The Cray SV1e has 32 hardware performance counters in four groups (0, 1, 2, 3) of 8 each. Only one group can be active at a time. The counters provide the user with valuable information on the execution of his code.

Group 0 : Number of

0 - Instructions issued
1 - Clock periods holding issue
2 - Instruction buffer fetches
3 - Floating-point add operations

4 - Floating-point multiply operations
5 - Floating-point reciprocal operations
6 - CPU port memory references
7 - Cache hits

Group 1 : Number of clock periods

0 - Waiting on A registers
1 - Waiting on S registers
2 - Waiting on V registers
3 - Waiting on B, T registers
4 - Waiting on Vector Functional Units
5 - Shared registers
6 - Waiting on memory ports
7 - Waiting on miscellaneous

Group 2 : Number of

0 - Instruction fetches
1 - Cache hits
2 - Scalar memory writes
3 - B, T memory references
4 - Scalar memory references
5 - CPU memory writes
6 - CPU memory references
7 - CPU memory conflicts

Group 3 : Number of

0 - 000-017 instructions
1 - 020-077 instructions
2 - 100-137 instructions
3 - 140-157 and 175 instructions
4 - 160-174 instructions
5 - 176 and 177 instructions
6 - Vector integer/logical operations
7 - Vector floating-point operations

Example of use

f90 prog.f
hpm -g0 ./a.out

See als**o man hpm**. There is virtually no overhead in using this tool.

3.6 Multi-Streaming Processor directives

Multistreaming is a feature that lets you schedule four dedicated Cray SV1e CPUs as one multistreaming processor (MSP). Multistreaming operates on the loop and array-syntax. Multistreaming on a Cray SV1e system is similar to Autotasking in distributing loop iterations across processors. However, multistreaming causes gang scheduling of all requested processors, meaning they are attached to the program whether they are actually executing code or not, so expect an increase in CPU time used. Multistreaming is an optional feature. The **sysconf** command on RuG-Cray indicates that zero multistreaming processors are configured. In the future this may change. You can alter this with **-O streamN,** N=0,1,2,3, as compiler directive.

Multistreaming directives

Directive	*Description*
!DIR$ STREAM	turns on feature where found in code.
!DIR$ NOSTREAM	turns off feature where found in code.
!DIR$ PREFERSTREAM	identifies next loop as the one to be MSP optimized when multiple candidates exist .

Scalar directives

Directive	*Description*
!DIR$ INTERCHANGE	applies to next loops, reorders nested loops.
!DIR$ AUXILIARY	(allocate to SSD as secondary memory) Requires existence of 'ex' memory which is configured as secondary memory.

Cache blocking directives

The word "blocking" does not refer to a barrier as in a blocking read. Rather, the compiler produces code to operate on blocks of the data space that fit into cache, in order to accomplish as much manipulation as possible on the block before bringing a different block of data into the cache.

Directive	*Description*
!DIR$ BLOCKABLE	following nested loops can be involved in cache blocking together.
!DIR$ BLOCKINGSIZE	applies to next loop; primary, secondary block size.
!DIR$ NOBLOCKING	next loop can't be cache blocked.
!DIR$ UNROLL n	0 <= n <= 1024.
!DIR$ CONCURRENT	number of safely concurrent loop iterations.

3.7 Network

The current and target networks connecting the Cray SV1e, the rest of the HPC&V compute and storage systems, and systems outside of the RuG are as follows.

Through a 100 Mbit/s Ethernet connection (service.rug.nl) the Cray SV1e is connected to other facilities of HPC&V such as the SGI Onyx 3400, the Intel cluster, the Alpha cluster and on the other side the RuG backbone (part of the national Surfnet) of 1 Gbit/s as a port to systems outside. On top, the local machines are coupled to the SAN (Storage Area Network) by 1Gb/s connections (at present partly realized).

Each machine can be reached (through FTP or Telnet) from anywhere, except from the SGI which will sometimes be used `stand-alone' in case of heavy duty real time visualizations.

3.8 I/O and File Systems

Disk drives, interfaces to other networks, and other peripherals are connected to the Cray SV1e using high-speed GigaRing (dual-rings) I/O channel.

- peak transfer rate per ring = 500MB/sec
- effective total bandwidth = 800MB/sec
- networking protocols used - ethernet
- cray.service.rug.nl IP address 129.125.50.9
- 250 Gbytes user disk space

3.9 Back up

Back up facilities are not realised yet! The partition **/u0** and **/u1** are configured with parity disks, such that in case of a disk crash files will not be lost. It is intended to add to the system in the near future a Tape Subsystem as back up facility.

3.10 Mass Storage

Mass storage facilities are not realised yet! It is intended to connect the Cray SV1e to the local SAN (Storage Area Network) of 25 Tbytes. Users can then copy their files to the SAN.
If the present situation causes problems, please notify HPC&V.

3.11 Performance

The peak performance of the SV1e is 64 Gflop/s. For a large number of problems the sustained performance is in the same order of magnitude, depending on the extent the code is vectorized/parallelized, cache/memory usage,etc.

4. Software

The Cray SV1e is a Unix machine (UNICOS version 10.0.1.0).

4.1 Programming Environment

To achieve maximum speed, a program must take advantage of the features of the SV1 architecture. Cray's compilers attempt to vectorize, parallelize and optimize programs, and can achieve significant speedup automatically. In many cases, however, the programmer must assist by manually restructuring some loops and I/O within the code. In some cases, the programmer will need to rewrite the entire program using appropriate algorithms, data structures, library calls, etc.

The *raison d'etre* of supercomputers is to solve problems that could not be solved on lesser machines (in a single life time or grant cycle, anyway). Thus, it is no surprise that Cray provides an arsenal of tools to help you create, analyze, vectorize, and optimize your codes.

Compilers

There are several compilers available, including: f90 (fortran 90), fort77 (fortran 77) , **cc** (C), **CC** (C++), and **as** (Cray assembler).

A few common options are given here for the f90 and cc compliers

FORTRAN compiling system**: f90**

f90 -o prog prog.f

Compiles, with the default level of optimization and vectorization, the FORTRAN program **prog.f**, producing the executable file, **prog**.

f90 -o prog -O3 prog.f

increases optimization level.

C compiling system: **cc**

cc -o prog prog.c

Compiles, with optimization and vectorization, the C program, **prog.c**, producing the executable file, **prog**.

cc -o prog -O3 prog.c

Ditto, but the **-O3** flag invokes a system to enhance vectorization and parallelization.

4.2 Fortran

The new f90 compiler options for SV1 as compared to those of the J90 are mainly:

Use **ftnlx** in preference to **ftnlist** or **ftnlint**.

no -Wp"srcpp_opt"	source preprocessor options
-J dir_name	directory for xxx.mod files
-M msgs	alter severity level of messages

-O shortcircuitN	faster evaluation of logical expressions
-O streamN	multistreaming processor
-r b	listing page breaks and headers
-r o	list compiler options used
-r t	list source code and lint style checking

enable options

-e d	recognize D debug lines
-e L	allow zero-trip shortloops
-e o	list optimization options on **stderr**
-e Q	accept variables with leading underscore
but no more	
-e X	generate code for ATExpert

4.3 Debuggers

Debuggers are the ultimate tool for determining why programs crash, behave strangely, or give incorrect output. They let you walk slowly through your code. You may execute one statement, N statements, one loop, or one subroutine, and then examine the contents of any (or all) arrays or simple variables before taking another step. If you still rely on print statements for large debugging chores, try the following tool

totalview: X Window System debugger

FORTRAN programs

f90 -g -o prog prog.f	Recompile your code with the **-g** flag
totalview prog	Run totalview on the executable

C programs

cc -g -o prog prog.c	Recompile your code with the **-g** flag
totalview prog	Run totalview on the executable

totalview: command line interface.

FORTRAN or C programs: Compile your program with the **-g** flag, as usual. Run totalview on the executable with the -L flag, which forces it to use an command-line interface e.g.,

totalview -L prog

Be sure to recompile your program without the **-g** flag before attempting production runs! This flag removes all vectorization and optimization, which ruins performance.

4.4 Performance Analysis Tools

The Cray SV1e is equipped with several performance analysis tools to help you determine the rate at which your program runs and to locate bottlenecks. Typically, you use them to find any heavily used but poorly vectorized sections of code, then rewrite these sections for dramatic overall speed-up.

Run Time Analysis

These tools measure the performance of your program in actual runs. If they indicate that a subroutine is particularly costly, then concentrate your optimization efforts on that routine.

ja
ja (job accounting) gathers and reports statistics on a collection of commands.
Initiate accounting with **ja**, terminate and report with **ja -cst** (note, the semi-colon separates multiple commands on the same line. You do not need to recompile).

f90 -o prog prog.f ; ja; ./prog ; ja –cst

hpm
hpm (hardware performance monitor) gathers statistical data on your program's use of registers and other hardware components and then prints a report including, for instance, Mflop/s and the number of floating point additions done.
Gather and print statistics on any executable (you do not need to recompile)

hpm ./any_executable

perfview
perfview provides command-line or X Window System interfaces to hpm statistics gathered on your program. It gives the time consumed by each subroutine, and offers friendly suggestions for speeding up your code.
Recompile your code with perfview flags, (-ef or-F) **-lperf**

f90 -o prog -ef -lperf prog.f FORTRAN

or

cc -o prog -F -lperf prog.c C

Run the program:

./prog

It will produce a perfview data file, **perf.data**
Run perfview

perfview X Window System interface - default

or

perfview -L Command line interface

To get started in perfview, try clicking in a wedge of the pie chart. To recreate the chart, go to the graph menu.

flowview
flowview provides a slightly different view of your program, again using **hpm** statistics. Its distinction is that it displays your program's actual run-time (dynamic) calling sequence.
Recompile your code with the flowview flag, (-ef or -F):

f90 -o prog -ef prog.f FORTRAN

or

cc -o prog -F prog.c C

Run the program. It will produce a flowview data file, **flow.data**

./prog

Run flowview. Check out all options in the graph menu

flowview	X Window System interface - default
or	
flowview -L	Command line interface

Remark
Timings can be a problem: total cpu or wall clock? Parallelization implies decreases in wall clock, not total cpu time. **flowview** in particular measures cpu time! Therefore tasking may make little difference to flowview. Use **atexpert** or **ja** to look for timings. Also look at overheads: small chunks may be well parallelized but overwhelmed by start-up code.

4.5 Libraries

The Cray SVe1 is a shared memory machine. This means, there is generally no need to explicitely move around data blocks in memory. If you do need it (e.g. you want your shared memory program to be compatible with distributed memory) then use MPI or SHMEM. See man-pages.

Optimized Libraries
The fastest code on the Cray SV1e is in the libraries of subroutines and functions provided by Cray. Many of these routines are written in assembly language, most run in parallel, and they all take maximum advantage of the SV1 vector architecture. For fast programs, use the library routines!

Cray libsci
Supported by Cray, with documentation available via **man**. To link in FORTRAN

f90 prog.f

(f90 loads libsci automatically if one of its routines is called.) To link in C

cc -lsci prog.c

(The flag, **-l**, causes the named library to be loaded.)

The Cray Scientific Library routines are part of the default programming environment on the Crays. See **man intro_libsci** for a summary.

The library includes:

Signal processing routines: see **man intro_fft**
Fast Fourier Transform (FFT) routines, filter routines, and convolution routines.

Solvers for dense linear systems and eigensystems: see **man intro_lapack**
The preferred solvers for dense linear systems are those parts of the LAPACK package included in the current version of the Cray Scientific Library. The LAPACK routines in the Scientific Library supersede the older LINPACK routines.

Vector-vector linear algebra subprograms: see **man intro_blas1**
The Level 1 BLAS perform basic vector-vector operations.

Matrix-vector linear algebra subprograms: see see **man intro_blas2**

The Level 2 Basic Linear Algebra Subprograms (Level 2 BLAS) consist of CAM or CAL routines for real and complex data. They handle matrix-vector operations.

Matrix-matrix linear algebra subprograms: see **man intro_blas3**
The Level 3 Basic Linear Algebra Subprograms (Level 3 BLAS) consist of routines for unpacked real and complex data. They handle matrix-matrix operations.

Solvers for special linear systems: see **man intro_spec**
All solvers for special linear systems run only on Cray PVP systems.

Solvers for sparse linear systems: see **man intro_sparse**
A sparse matrix is a matrix that has relatively few nonzero values. This type of matrix occurs frequently in key computational steps of a variety of engineering and scientific applications. Most sparse matrix software takes advantage of this "sparseness" to reduce the amount of storage and arithmetic required by keeping track of only the nonzero entries in the matrix.

Out-of-core routines: see **man intro_core**
The Cray Research Scientific Library out-of-core routines for linear algebra let you solve problems in which it is not possible, or not convenient, to store all of the data in main memory during program execution. The central concept on which these routines are based is the idea of the virtual matrix, which is stored outside main memory, and referenced through a Fortran I/O unit number.

Machine constant functions: see **man intro_mach**
These functions return machine constants for Cray Research systems.

NAG Library Contents
The NAG Fortran library is third party software and contains over a thousand numerical routines covering almost each numerical problem. The library is optimised (vectiorised/parallellised) in the following sense. Routines are essentially sequential. If the routine calls a BLAS (Basic Linear Algebra System) routine then this Blas-routine may run in parallel.

A complete description of the routines is found at

www.nag.co.uk/numeric/manual/htm/genint/Fllibconts.asp

The library is linked with **–lnag.**

The chapter layout of NAG is

A00	Library Identification
A02	Complex Arithmetic
C02	Zeros of Polynomials
C05	Roots of One or More Transcendental Equations
C06	Summation of Series
D01	Quadrature
D02	Ordinary Differential Equations
D03	Partial Differential Equations
D04	Numerical Differentiation

D05	Integral Equations
D06	Mesh Generation
E01	Interpolation
E02	Curve and Surface Fitting
E04	Minimizing or Maximizing a Function
F01	Matrix Factorizations
F02	Eigenvalues and Eigenvectors
F03	Determinants
F04	Simultaneous Linear Equations
F05	Orthogonalisation
F06	Linear Algebra Support Routines
F07	Linear Equations (LAPACK)
F08	Least-squares and Eigenvalue Problems (LAPACK) A list of the LAPACK equivalent names is included
F11	Sparse Linear Algebra
G01	Simple Calculations on Statistical Data
G02	Correlation and Regression Analysis
G03	Multivariate Methods
G04	Analysis of Variance
G05	Random Number Generators
G07	Univariate Estimation
G08	Nonparametric Statistics
G10	Smoothing in Statistics
G11	Contingency Table Analysis
G12	Survival Analysis
G13	Time Series Analysis
H	Operations Research
M01	Sorting
P01	Error Trapping
S	Approximations of Special Functions
X01	Mathematical Constants
X02	Machine Constants
X04	Input/Output Utilities

Next to the above described numerical libraries, visualization software like KOMPLOT (library for 2D graphs) and AVS (advantaged 3D visualization package) are available. See www.rug.nl/hpc/hp/cray.htm. If you need more specialized libraries, please contact HPC&V.

4.6 Documentation

A general way to obtain a quick overview of all available introduction man-pages is **man –k intro**.
Next to referencing the **man** pages**,** you may wish to use the online Cray manuals (which supersede **docview**). There are a large number of manuals available as HTML, PostScript or PDF files on Cray's online document site

http:www.cray.com/craydoc/

If you are browsing on this site, note that the newest documents are those with ID formats like S-2312-35 (35 for Programming Environment 3.5) rather than the Cray Ids with two leading characters like SG-2192 3.0 or SGI Ids with three leading digits like 004-2192-003.

Some relevant documents are

Number	Title
S-3695-35	Application Programmer's I/O Guide
S-3901-35	CF90 Commands and Directives Reference Manual
S-2312-35	Cray SV1 Application Optimization Guide
S-3692-35	Fortran Language Reference Manual, Volume 1
S-3693-35	Fortran Language Reference Manual, Volume 2
S-3694-35	Fortran Language Reference Manual, Volume 3
004-2179-004	Cray Standard C and C++ Reference Manual

Note: The Cray SVe1 is a PVP architecture running UNICOS operating system. In the documentation also other architectures and operating systems are frequently mentioned (e.g. UNICOS/mk and Cray T3E). You can safely skip that information.

5. Optimizing

If you aren't very much interested in optimizing your code (for instance if you run only once) then you may want to skip this section and leave optimization completely to the compiler, i.e. use **–On** in the compiler option **(**consult the **man** page of the compiler) and consult the source listing, e.g. **f90 –r3 prog.f** results in a file **prog.lst** containing the source code with parallel or vector messages, cross-references , etc.

5.1 Basic Concepts

The Cray SV1e is a shared memory machine with 32 processors. There is a big central pool of memory, and each processor can access this memory via a crossbar. Cray puts considerable effort into designing a very fast, and intelligent, crossbar which can feed the processors very efficiently. (In a distributed memory machine such as HPC&V's Alpha and Intel cluster each processor would have its own memory).
Unix (and UNICOS) is quite naturally able to exploit parallelism because each task in Unix is a process which has its own private chunk of memory. Processes can be put in the background and run concurrently (in parallel) on different processors.

Processes

- Each process is completely independent: no special programming required.
- Each process is internally a single processor task. UNICOS may run a process over many processors. Execution is, however, ordered (scalar). Memory is kept separate from other processes.
- Not so good if data from some other process is required. Mechanisms to send and receive data are available in Unix, it is rather expensive as they require system calls and memory copies.

5.2 Cray Multi-tasking

A generic term for a process which utilizes many processors. May be explicit with calls to a parallel processing library (threads, MPI). Or implicitly produced by the compiler (dataparallel). In this introduction we will concentrate on automatic compiler parallelization. Cray calls this tasking. For an example of MPI and SHMEM calls, see Section 6.7.

Drawbacks

Before going any further here are a few potential problems

- Generally requires quite a lot of programmer effort
- Can be fiendishly difficult to debug.
- Many different ways to parallelize.
- Introduces computer overheads:
    - More memory: usually for temporary workspaces on each processor.
    - Inter-processor communication.
    - wait-time as processors wait for others to finish.

Therefore

- You must really need the speed!

- Algorithm should be naturally parallelizable.

Note on PVM
PVM is the Parallel Virtual Machine library. It is designed to allow independent processes to send messages (data) between themselves. It has the great advantage of working on dissimilar processors, and can also send messages over a network between different computers.
But clearly it is aimed for distributed memory systems and has no functionality for shared memory, and is therefore not suited for the Cray SV1 machine. Cray supplies PVM for the SV1 so that it can perform as a single "node" in a network of other machines (in particular a network of Cray supercomputers). They are not supplied for parallelization within the SV1 itself.

5.3 Parallelization Levels

There is quite a lot of stuff to be familiar with before coding. First Cray splits code parallelization into three levels:

1. Low Level: processor internals

- Single processor: all modern processors have some built-in parallelism: e.g. add and multiply pipelines. Usually considered to be "scalar" optimization.
- Cray has "vectorization". (This requires hand-optimization of assembly code: let the compiler do it.)

2. Intermediate Level: Loops

Parallelization of loops

Fortran 77 example

```
do i = 1, m
        x(i) = a(i) + b(i) * c(i)
end do
```

Fortran 90 example

```
X = A + B * C
```

C++ example

```
for( int i=1; i<=n; ++i ){   // each i on a separate processor
        v1[i] = v2[i] + v3[i];
}
```

The compiler splits this into chunks. Each chunk is vectorized and the chunks are done in parallel on different processors.

3. High Level: Different Subprograms
Different subprograms run on different processors. Very difficult because the compiler has to check for data dependencies between two sub-programs.
May require explicit programming (or at least special compiler calls) to ensure synchronization.

```
call sub1( )  ! on processor 1
call sub2( )  ! on processor 2 (is it really independent??)
```

Cray Approaches to Parallelization
Cray calls this "tasking". There are three basic methods

1.Macrotasking

- Different subprograms run on different processors.
- Independent subprograms explicitly identified by programmer.
- Library Routine calls: Not portable
- It works well only on programs with large granularity parallelism.
- ANSI standard Threads library is a more-or-less portable replacement.

2.Microtasking

- Code in loops runs on different processors.
- Parallel loops identified by compiler directives.
- Compiler driven: portable (will run on single processor)!

3.Autotasking

- Autotasking is the automatic detection and usage of multiple CPUs for a Fortran 90 or C++ program.
- Loop level again
- Parallel loops identified automatically by compiler.
- (May still require microtasking directives).

Comments

Macrotasking and Threads
The first option, Macrotasking, is becoming a standard approach to shared memory multi-processors because of the prevalence of such machines at the "departmental server" level. The threads library is a response to the need for Unix system programming on such a system. Thread programming can get quite complicated, but gives essentially all levels of control to the programmer. Cray inhouse routines are more efficient than threads implementations, but threads libraries are now universally available (at least three public domain implementations for Linux for instance).

Threads or similar libraries require calls to the operating system to set themselves up (find a processor, grab memory, hook into resources, ...). These operations are quite expensive so a few larger threads are to be recommended. Threads are therefore well-suited to "coarse-grained" parallelism.
Since threads really requires a book in itself we will leave the topic in favour of the automatic methods.

Microtasking and Autotasking
Fine-grained parallelism requires that the compiler directly generate machine code so that many, small parallel regions can be executed without the associated thread overhead. This is the origin of the "microtasking" approach.

Autotasking is really an outgrowth of Microtasking in that parallel regions can now be automatically identified if they are simple enough. If they are not recognized then explicit compiler directives from "Microtasking" can be given.
Since the compiler directly generates assembly code this approach can be considerably more efficient, and is suited both for loop and subroutine-level parallelism.

However, Autotasking can be more efficient than microtasking because they use rather different compiler approaches to generating code

- Microtasking: whole subprogram runs on many processors
- Autotasking: master process spawns slave processes as required.

Microtasking runs in subprogram scope. A subprogram grabs some number of processors and everything in the subprogram is run. This can lead to inefficiencies as scalar sections of code result in active but non-productive processors.
Autotasking is much better in that the code runs on a single master processor. If a parallel region is entered and there are processors available then the master can grab these processors and tell them what to do. On exit from the parallel region the slaves are relinquished and other processes can get to them.

5.4 Parallel processing terminology

Multitasking	To execute different parts of the program simultaneously using multiple processors.
Synchronization	Bringing two or more processes to known points in their execution at the same clock time.
Overhead	Extra work which must be performed by programs executing on multiple CPUs, extra time spent waiting at synchronization points for work to complete on other CPUs.
Granularity	A description of the amount of work given to a processor at run time. Fine-grain parallelism is often exploited at instruction or loop levels. Coarse-grain parallelism is exploited at the subprogram levels. In general, the finer the grain size, the higher the potential for parallelism and the higher the communication and scheduling overhead. Fine grain provides a higher degree of parallelism, but heavier communication overhead, as compared with coarse-grain computations.
Parallel region	Region of code which is executed by multiple processors.
Partitioned code	Code within a parallel region in which multiple processors share the work that needs to be done (make sure about data dependency!).
Redundant code	Code within a parallel region in which processors duplicate the work that needs to be available to all processors.
Data Scoping	The process of identifying the data items that are referenced within a parallel region. Each CPU is allocated a separate copy of *private* data. There is only one single storage location for *shared* data. The following rules determine shared or private status of different variables: **Shared** • Variables or arrays in a SHARED parameter. • Variables or arrays that are read only. • Arrays indexed by the loop index. • Variables or arrays that are read-then-write.

	Private • Variables or arrays in a PRIVATE parameter. • Variables or arrays that are write-then-read.
Critical region	Region of parallel code that must be executed by only one CPU at a time.

General Method

1. Get the syntax right. Compile and run test code.
2. Decide on a few test cases.
3. Compile and run with profiling turned on. Identify subroutines/regions which require work. Simple optimization: split loops, look for i/o, ...
4. Standard library vectorization/parallelization. BLAS routines are 2-4 times faster than compiler. Look for linear algebra: explicitly replace by BLAS. Consider the use of libraries like NAG.
5. Compile with vectorization and listings turned on. No profiling (high overheads, "-g") Turn on flowview for subprogram level profiling. Find time-consuming subprograms. Look at listings of those slow subprograms. In particular look for "not vectorized" loops. Fix loops either by re-coding or by explicit compiler directives (apply vectorization techniques).Test again
6. Compile with parallelization (and vectorization, listing). Same approach as for vectorization: No profiling (high overheads, "-g"). Turn on flowview for subprogram level profiling. Find time-consuming subprograms. Look at listings of those slow subprograms. In particular look for "not tasked because..." messages. Fix loops either by re-coding or by explicit compiler directives. Test again.
7. Consider advanced parallelization. Sets of subprograms to be farmed out: compiler directives. Generally this does not help much because farmed out subprograms need to talk to each other. Re-code using threads, MPI, etc.

5.5 Compiler Autotasking

Turn on autotasking with Cray's f90 and C++ compilers as follows

```
f90 -O task3 -r3 program.f90
CC -h task3 -h report=fisvt program.C
```

Reporting is turned on here to give info on compiler autotasking. There are four levels of tasking:

task0	none
task1	only directives (no automatic parallelization)
task2	some
task3	lots (agressive)

Number of Processors

Additionally the user can control the maximum number of processors that the system will allow the program to use, e.g.

```
csh> setenv NPCUS 32
```

5.6 Parallel processing directives

If the compiler does not recognize the parallel regions existing in your program you might want to have more control over different portions of your code. You can do so with parallel processing compiler directives.

General format

Fortran

```
!MIC$       directive parameters
!MIC$+      parameters continued
```

C

```
#pragma _CRI directive parameters \
parameters continued
```

Example: Matrix Row Norm

This is a standard example. Compiler directives are used to identify parallel loops, and to detail the behaviour of memory in these loop.
Approach: each row-sum is computed independently (in parallel). The result stored in a temporary vector. This gives a vectorized sum loop for each processor (the sum along the row). The maximum is computed by breaking the standard search loop into segments. Each processor finds the maximum in its segment. Finally, the segment maxima are compared (uniprocessor) to find the norm.

double row_norm(double uold, double** unew, const int nrow, const int ncol)**

```
// Norm
{
	// Temporary to hold sum across rows
	double* row_sum = new double[ncol+1];
	// Compute row sums in parallel
#pragma _CRI parallel shared( row_sum, unew, uold, nrow, ncol )
#pragma _CRI taskloop
	for( int irow=1; irow<=nrow; ++irow ){
		double sum = 0.0;
		for( int icol=1; icol<=ncol; ++icol ){  // vectorized
			sum += fabs( unew[irow][icol] - uold[irow][icol] );
	}
	row_sum[irow] = sum;
}

#pragma _CRI endparallel


// Compute max of row_sums in parallel segments
const int n_segments = 4;    // 4 processors
const int stride = 1 + (nrow-1) / n_segments;
double max_element[n_segments];

#pragma _CRI parallel shared( row_sum, max_element, nrow, ncol, stride, n_segments )
#pragma _CRI taskloop

for( int i_seg=0; i_seg<n_segments; ++i_seg ){
	const int i_start = i_seg * stride;
```

```
        int i_end = i_start + stride;
        if( i_end > nrow-1 ) i_end = nrow-1;
        double max_wrk = row_sum[i_start];
        for( int irow=i_start; irow<=i_end; ++irow ){  // not vectorized: recurrence
                if( row_sum[irow] > max_wrk ) max_wrk = row_sum[irow];
        }
        max_element[i_seg] = max_wrk;
}
#pragma _CRI endparallel

// Find maximum of the segments
double max_wrk = max_element[0];
for( int i_seg=0; i_seg<n_segments; ++i_seg ){
        if( max_wrk < max_element[i_seg] ) max_wrk = max_element[i_seg];
        }
delete[] row_sum;
return max_wrk;
}
```

Notes

- Automatic (no directives) gave a speed-up of 1.2 for 4 processors.
- The code in the basic block "parallel, endparallel" can be tasked.
- The parallel directive also details local or shared memory:
- **#pragma _CRI parallel shared(row_sum, unew, uold, nrow, ncol)**
- The given variables are shared by the processors, i.e. each processor accesses the same memory area.
- Memory can also be "private":
  #pragma _CRI parallel private(i) shared(row_sum, unew, uold, nrow, ncol)
  The variable i is private to each processor, i.e. each processor will have its own independent copy.
  C++ is useful here because variables can have local scope:
  ```
  #pragma _CRI parallel shared( row_sum, unew, uold, nrow, ncol )
          for( int irow=1; irow<=nrow; ++irow ){
                  double sum = 0.0;
          ...
  ```
  i and sum are automatically local (or private) because they are declared after the #pragma. Note that Fortran and C require that all variables have subprogram scope and things like counter must be declared private.
- Loops identified by
  #pragma _CRI taskloop
- Nasty debugging problems: Be very careful with shared and private data. Unpredictable (random) behaviour occurs, for instance, if a loop counter is shared between processors.
- Always carefully check data for hidden dependencies. The compiler does NOT do this.

Fortran Autotasking Compiler Directives

Similar directives exist for Fortran:

```
!mic$ parallel shared( row_sum, unew, uold, nrow, ncol )
!mic$2 private( irow, icol, sum )
!mic$ do parallel
```

```
      do irow=1, nrow
            sum = 0.0
            do icol=1,ncol
                  sum = sum + abs( unew(irow,icol) - uold(irow,icol) )
            enddo
      row_sum(irow) = sum
      enddo

!mic$ endparallel
```

Notes

- The directives are of the form "!mic$ .."
- Lines can be continued with line number after the "!mic$"
- All variables are declared as either shared or private.
- This particular code may be inefficient because storage is wrong. Fortran stores down rows; it would be better to use the column-norm, and parallelize on columns.

More Fortran Examples

```
!  In this example, a parallel region is defined by PARALLEL and END PARALLEL.

      sum = 0.0
      big  = -1.0
!MIC$        PARALLEL  PRIVATE (XSUM, XBIG, I)
!MIC$1       SHARED (SUM, BIG, AA, BB, CC)
      xsum = 0.0
      xbig = -1.0
!MIC$        DO PARALLEL
      do i = 1, 2000
            :
            xsum = xsum + (aa(i) * (bb(i) - cc(z(i))))
            xbig   = max (abs(aa(i) * bb(i)), xbig)
            :
      end do

!  GUARD/END GUARD protects the updating of shared variables SUM and BIG.

!MIC$   guard
      sum = sum + xsum
      big   = max (xbig, big)
!MIC$   end guard
!MIC$   end do
!MIC    end parallel
```

Example of F77 code (while using the F90 compiler)

```
      parameter (n=100)
      common /blkl/ a(n,n), b(n,n), c(n,n)
c
```

```
      do 200 j=1,n
         do 100 i=1,n
            a(i,j) = 0.0
            b(i,j) = i*.7 + j*.3
            c(i,j) = i*.9 - j*.2
 100  continue
 200  continue
      ipnum = 1

!MIC$ PARALLEL SHARED(a, b, c, ipnum) PRIVATE(i, j, k, kold, mypnum)
!MIC$ GUARD
      mypnum = ipnum
      ipnum = ipnum + 1
!MIC$ END GUARD
      kold = 0
!MIC$ DO PARALLEL
      do 600 k=1,n
      write(6,*) 'Process ', mypnum,': Ending: ',kold, '  Starting: ',k
 1    do 400 j=1,n
         do 500 i=1,n
            a(i,k) = a(i,k) + b(i,j) * c(j,k)
 500     continue
 400  continue
      if( ranf() .lt. .99 ) goto 1
      kold = k
 600  continue
!MIC$ END PARALLEL
      stop
      end
```

F90 Example with an artificial quirk
Observe the "big_sum" input parameter used to perform a global sum. Difficult, because multiple processors shouldn't be updating it simultaneously.

```
function test_norm( a, weight, nrow, ncol, big_sum ) result( out )
!
! Modified weighted norm with artificial "big_sum" for multiple sums from main program.
! This version has explicit parallelizations.

      use sysconst
      use pconst

      integer, intent(in) :: nrow, ncol
      real (kind=BIT64), dimension(nrow,ncol), intent(in) :: a
      real (kind=BIT64), dimension(ncol), intent(in) :: weight
      real (kind=BIT64), intent(inout) :: big_sum
      real (kind=BIT64) :: out
      real (kind=BIT64) :: col_sum, norm_sum
      integer irow, icol
```

```
        norm_sum = 0.0

!MIC$         PARALLEL SHARED(a,weight,nrow,ncol,norm_sum,big_sum)
!MIC$2        PRIVATE(irow,icol,col_sum)
!MIC$         DO PARALLEL

        do icol=1, ncol
                col_sum = 0.0;
                do irow=1, nrow
                        col_sum = col_sum + a(irow,icol)*a(icol,irow)
                end do
                norm_sum = norm_sum + col_sum
        enddo

!MIC$         guard
        big_sum = big_sum + norm_sum*0.5
!MIC$         endguard

!MIC$ END DO
!MIC$ END PARALLEL

        out = sqrt( norm_sum )
end function
```

- Note the "!MIC$ guard" region.
- Precludes the following

    Processor 1 retrieves BIG_SUM
    Processor 2 retrieves BIG_SUM
    Processor 1 performs add.
    Processor 2 performs add.
    Processor 2 puts BIG_SUM
    Processor 1 puts BIG_SUM
    Result: Processor 2 contribution is lost.

Example Parallel Subprograms

This is very simple: use a "!MIC$ CASE"

```
...
!MIC$         PARALLEL SHARED(...)  ! set up parallel region
!MIC$2        PRIVATE(...)

!MIC$         CASE               ! give cases
        call func1()
!MIC$         CASE
        call func2()
!MIC$         CASE
        call func3()
!MIC$         ENDCASE
!MIC$ END PARALLEL
```

...
Note: these subroutines can't talk to each other. You have to use threads if more is required.

Parallel processing performance issues
Multitasking provides a means to reduce the wall-clock execution time for a program relative to single processor execution. With N CPUs, a speed up ratio equals to N is desired. But there are some limitations to achieve this speed up for a multitasked code. They are of course the maximum number of available processors (**p**) and the fraction of your code that can be run in parallel (**f**). This speed up can be expressed by Amdahl's law. Amdahl's law calculates the theoretical maximum speed up (**S**) for a program: $\mathbf{S = 1/((1 - f) + f / p)}$

		p	4	8	16	32	1.0e6
f	.1		1.08	1.10	1.1	1.1	1.1
	.25		1.23	1.28	1.3	1.3	1.3
	.4		1.43	1.54	1.6	1.6	1.7
	.5		1.60	1.78	1.9	1.9	2.0
	.6		1.82	2.11	2.3	2.4	2.5
	.7		2.11	2.58	2.9	3.1	3.3
	.8		2.50	3.33	4.0	4.4	5.0
	.9		3.08	4.71	6.4	7.8	10.0
	.95		3.48	5.93	9.1	12.5	20.0
	.975		3.72	6.81	11.6	18.0	40.0
	.98		3.77	7.02	12.3	19.8	50.0
	.99		3.88	7.48	13.9	24.4	100.0
	.995		3.94	7.73	14.9	27.7	200.0

Maximum Theoretical Speedup on **p** CPUs with fraction **f** of parallelism.

5.7 ATExpert

ATExpert is a tool developed to measure and graphically display the autotasking performance of a job run on an arbitrarilly loaded Cray research system. ATExpert provides accurate performance measurements of a program, subroutines, parallel regions, and loops in a given application. Times for starting, stopping, and performing parallel work in a parallel region are measured. ATExpert also measures the time to perform work between parallel regions (called preceding serial time) and following the last parallel region (called ending serial time).
ATExpert addresses two key issues:

- What improvement did using the Autotasking option give me?
- What can I do to improve it?

Compile with a suitable flag, run the program, and then run the **atexpert** utility.

Fortran 90

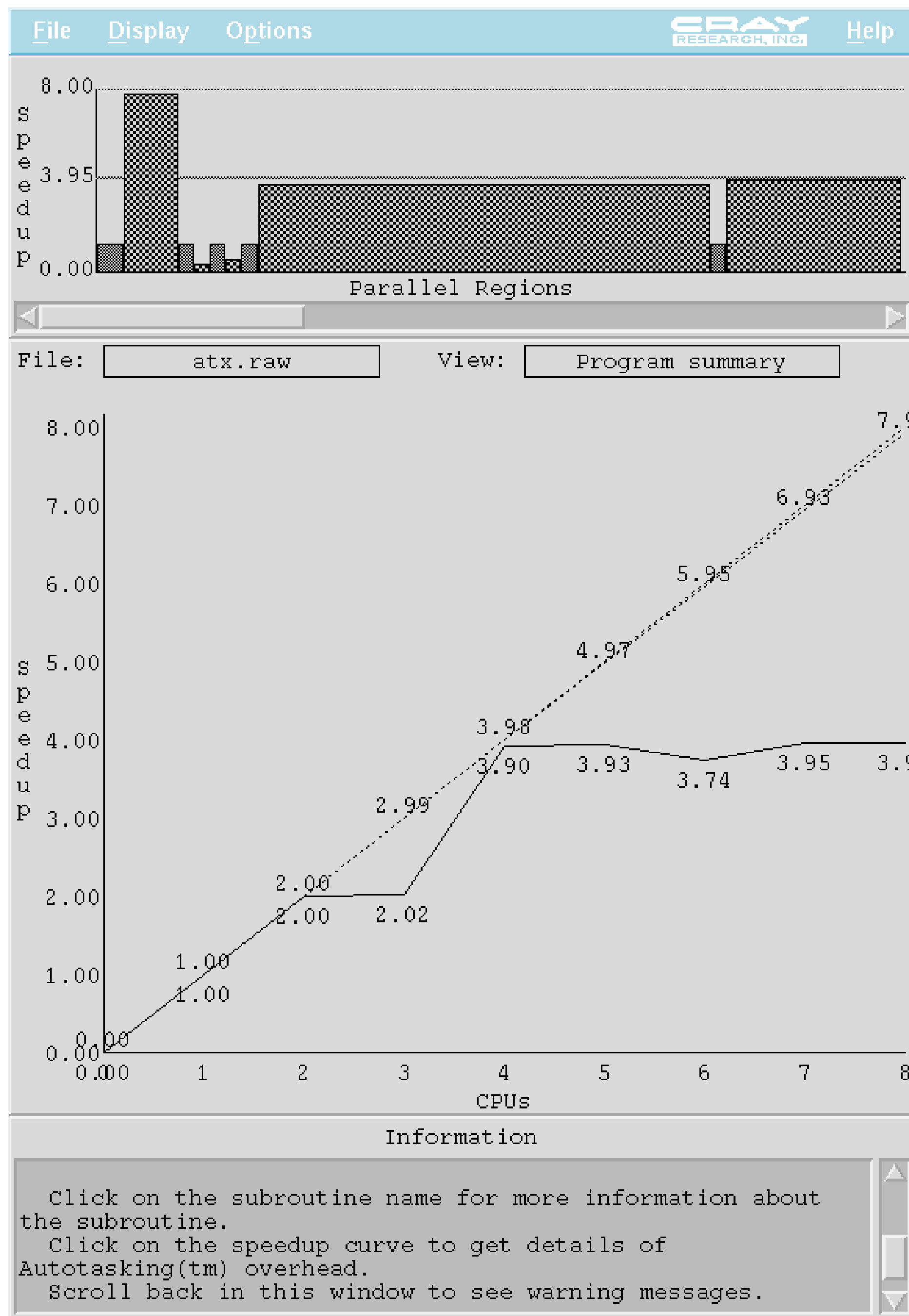
```
f90 -eX $(sources) -Otask3 -o main
./main
setenv DISPLAY machine_name:0
atexpert
```

C++

```
CC -hatexpert -htask3 $(sources) -o main
./main
setenv DISPLAY machine_name:0
```

atexpert Windows

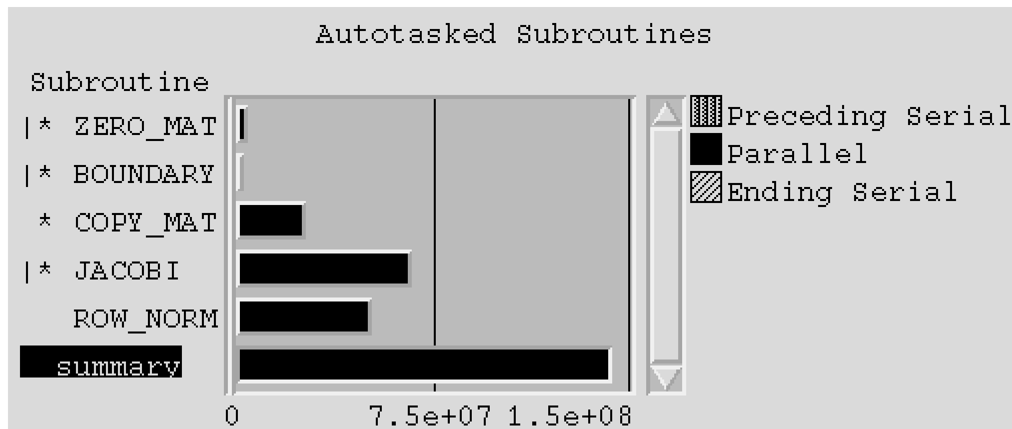
The main window is a graph of the performance of the code as a whole:



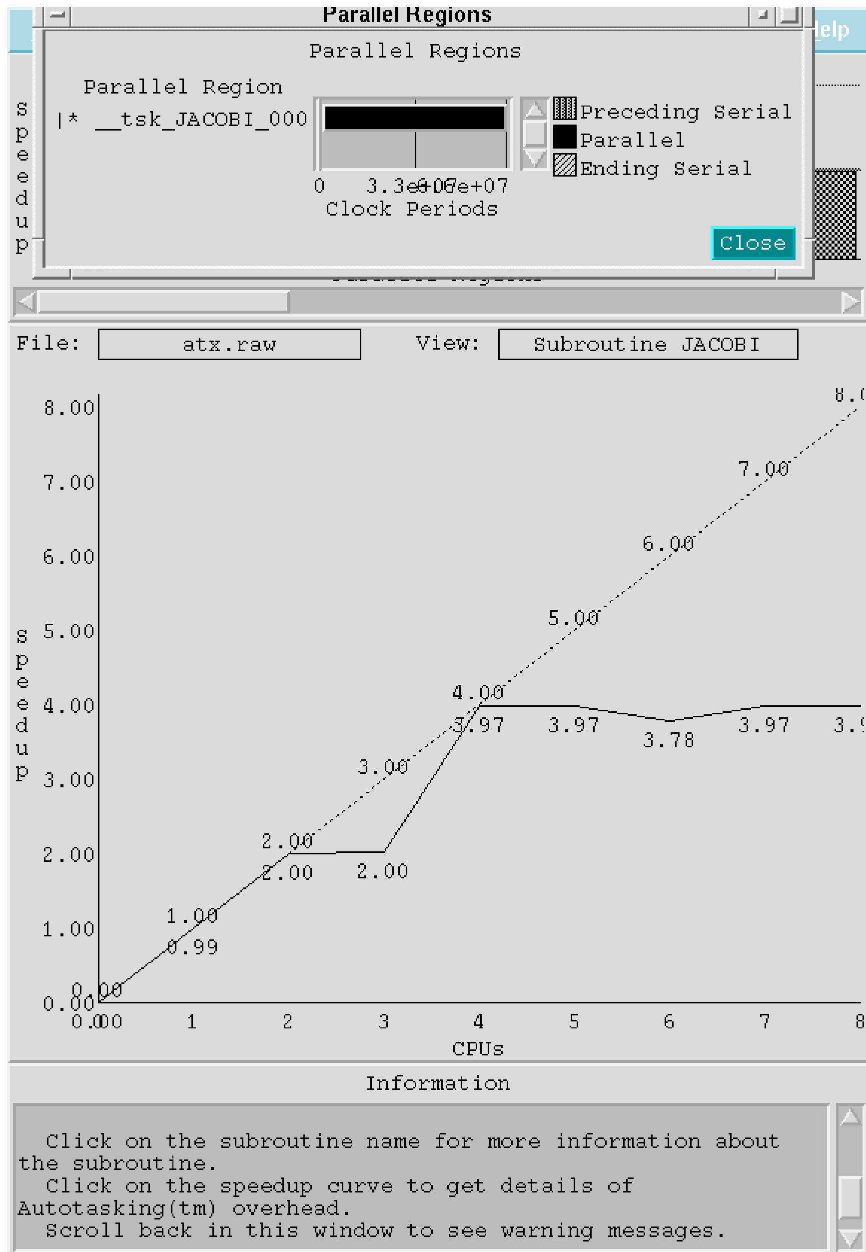
Diagonal line corresponds to the maximum linear speed up. Dashed line is theoretical maximum speedup using Amdahl's law. Solid line is the prediction based on the execution profile. This particular routine clearly parallelizes up to NCPUS=4!

Autotasked Subroutines Window

Another window shows what function is being analyzed. Click on the function name to get details about that particular function.

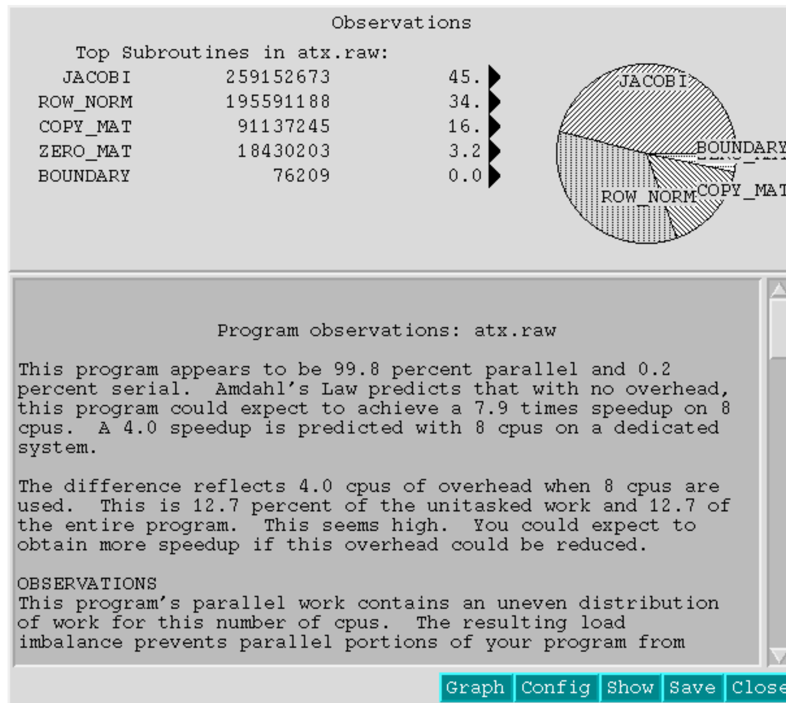


The summary entry is highlighted. Preceding Serial: Indicates the time from when the previous parallel region ended to the time that the next parallel region begins. Parallel: Indicates the sum of parallel times of all parallel regions in this subroutine or function.
Clicking on the JACOBI entry gives, for instance:

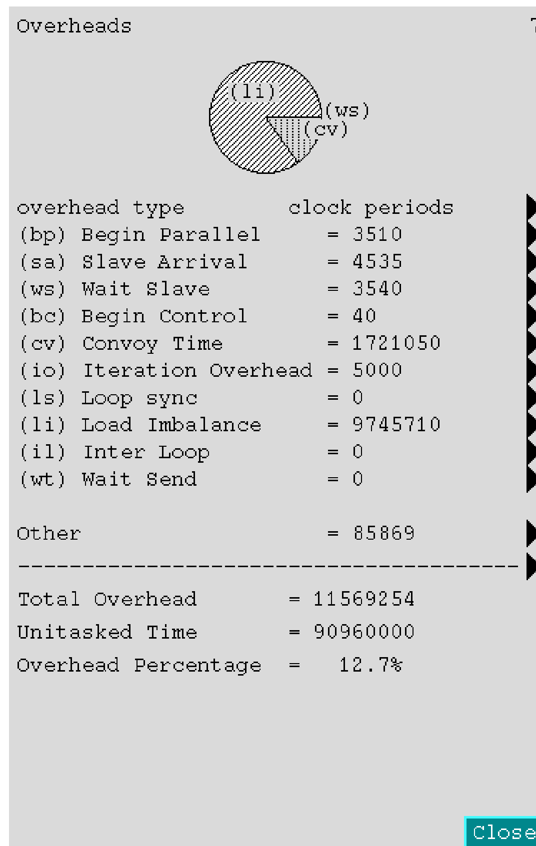


atexpert Parallelization Hints

Click on the menu items to get some hints about further parallelization, e.g.



or



Remarks

- Generally, the programmer makes cross-references between **atexpert** and compiler listings.

- Compiler listings are still the ones that best tell why a loop has not been vectorized or parallelized.
- **atexpert** indicates which loops or subroutines to concentrate on.
- Libraries, that are microtasked or autotasked without **atexpert** instrumentation, will be measured as preceding serial time. This includes **libsci.**

Conclusions
Parallelization via autotasking and compiler directives is simple:

- Directives are reasonably straightforward.
- Good compiler listings and tools to analyze result.

It falls apart when individual subprograms are run in parallel and need to talk to each other. Use threads.

6. Support

6.1 Training

Training of new users of the Cray SV1e can be provided. Training may contain Unix, C, C++, vectorisation, Parallellisation and MPI, depending on the user's request.
From time to time an EC summer school on parallelization is organised.
Special issues such as Multi-streaming and logical expression acceleration (important to e.g. genomics) will be addressed in separate short sessions.

6.2 Consultancy

HPC&V consults on all matters of use of the Cray SV1e, e.g. on hardware, algorithms, system software, compilers and the kind. Consultancy has no financial consequences if limited to a few days.
The Helpdesk function is via email and concerns only short well-defined questions.

6.3 Projects

HPC&V is experienced in collaborative projects. These projects vary largely in financial magnitude, scientific and technical contents, number of groups associated, way of subsidising (EC, NWO, etc.) . The main issues, as far as HPC&V is involved, are in supplying technical facilities, designing new or vectorized/parallelized algoritms, dessimination of knowledge, etc. HPC&V has been a partner in many local, national and international succesful projects, and may be yours in the future.

HPC&V staff engages in many research and development activities. Most staff research and development projects are collaborations with other researchers. We are interested in partnering with computational researchers in which HPC&V computational expertise can enhance the research efforts. We are also interested in partnering with computer scientists, technologists, and developers to develop new tools and techniques. Finally, we are interested in collaborations to develop proposals for joint hardware and software acquisitions. If you are interested in partnering with HPC&V on a proposal or an ongoing research/development project, please contact us

Visiting address	Nettelbosje 1, 9747AE, Groningen, The Netherlands
P.O. Box	11044, 9700CA Groningen, The Netherlands
Mail	*HPCV@rc.rug.nl*
Telephone	+31 50 3638080
Fax	+31 50 3633406
Internet	www.rug.nl/hpc

Appendix A

Bit Matrix Multiply

Since Bit Matrix Multiply is an especially useful new hardware function of the SV1e, we will pay some attention to it. It can accelerate your code dramatically if this technique is applicable. At the end of this section an example is given where two matrices are multiplied: in bit-representation the speed-up is a factor thousand (1000!) as compared to an integer-representation (which is already fast: 1.7 Gflop/s on one CPU).

Description
The Bit Matrix Multiply (BMM) hardware functional unit is standard on all CRAY SV1 systems. It can be used to perform very fast matrix multiply operations of two bit-matrices.
Because the BMM intrinsic functions generally map directly onto the hardware, there are additional restrictions placed on their use beyond language conventions, and some additional care must be taken when writing code that uses them. Most BMM functions must be vectorized in order to generate correct code. The vector version of this intrinsic is used when **-h vector3** (C compiler) or -**O vector3** or **-O3** (Fortran compiler) has been specified on the compiler command line.
The following section describes the individual routines for Fortran (for C/C++ see **man bmm**).

The BMM intrinsic functions are transformational functions. The names of these intrinsics cannot be passed as arguments. They accept the following arguments

x, y — Array or scalar values that are left-justified, zero-filled, and up to 64 bits in width. In Fortran, they should be INTEGER (KIND=8) values.

The following is a summary of BMM functions

M@LD — Load transposed bit matrix
M@LD loads the BMM functional unit with a matrix in transposed form. The function must appear in a vector loop or in an array syntax statement of 64 or fewer iterations, or an error message is issued. The SHORTLOOP compiler directive may be used to indicate a loop with between 1 and 64 iterations.
Use of array syntax for M@LD, as shown in the following example, is preferred because most simple array syntax vectorizes independently of the optimization level

BMM_STUFF(1:64) = M@LD(IA(1:64))

M@MX — Perform bit matrix multiply
M@MX performs the bit matrix multiply of its argument times the transpose of the argument previously loaded into the BMM functional unit with a M@LD, M@LDMX, or a two-argument M@MX invocation. With one argument, M@MX may appear in a scalar context. This intrinsic function can run at the maximum vector length supported by the hardware.
The 2-argument form of M@MX is functionally identical to the M@LDMX intrinsic function. It abides by the same rules and restrictions. Basically, the 2-argument form is equivalent to using M@LD followed by a 1-argument M@MX. Use of the 2-argument form of M@MX is discouraged; use M@LDMX instead. Use of array syntax for M@MX is preferred.
The following is a common example of performing a bit matrix multiply:

```
BMM_STUFF(1:64) = M@LD( IA(1:64) )          ! Load BMM unit transpose
IRESULT(1:512)  = M@MX( IB(1:512) )         ! Multiply times IB
```

M@LDMX Load bit matrix and multiply

M@LDMX combines the load and multiply functions and is functionally identical to the 2-argument form of M@MX. It must appear in a vector loop or in an array syntax statement, or an error message is issued. The second argument is loaded into the BMM functional unit, just as if M@LD was used, and is multiplied by the first argument.

Because of the load into the BMM functional unit, loops containing this intrinsic function are run with a maximum vector length of 64. The use of array syntax for M@LDMX, as shown in the following example, is preferred

```
IRESULT(1:512) = M@LDMX( IB(1:512), IA(1:512) )
```

M@UL Unload bit matrix

M@UL unloads the BMM functional unit and returns the transpose of the vector initially loaded into it by a M@LD or M@LDMX intrinsic. This is done by multiplying the unit's contents by an identity matrix. This intrinsic function must appear in a vectorizable loop or in an array syntax statement of 64 or fewer iterations, or an error message is issued.

The use of array syntax for M@UL, as shown in the following example, is preferred

```
ITRANS(1:64) = M@UL( )
```

M@CLR Clear Bit Matrix Loaded (BML) bit

On the system, M@CLR clears the exchange packets and the BML bit of the CPU status register. This is useful for reducing system overhead when the bit matrix portion of a code is inactive. If the BML bit is set, the operating system saves and restores the contents of the BMM functional unit when switching from user to user. Clearing this bit with M@CLR when you no longer are using the BMM functional unit allows the operating system to omit the save and restore steps. The value returned by this intrinsic can be safely ignored. Example:

```
IT = M@CLR( )
```

Remark

Because the BMM functional unit has a memory, you should be aware of the conventions needed to manage it. The UNICOS operating system maintains the BMM functional unit's memory when disconnecting and connecting users from CPUs. The contents of the BMM unit is undefined upon entrance to a procedure or function and undefined after calls to non-intrinsic functions or procedures. To preserve the contents of the BMM functional unit across function or subroutine calls, the only completely safe method is to unload and save the contents with the M@UL intrinsic and restore the contents of the functional units before an intrinsic is used.

Examples

The array syntax used for the bit matrix operations in this example is simple enough to vectorize even with optimization disabled.
This simple example demonstrates a bit matrix multiply. The bit matrices used are similar to the following, but those used are actually larger. Multiply

```
1111        1000        0101
0111   x    1100   =    1101
0011        1110        0001
0001        1111        1111
```

A program that uses the BMM routines is as follows

```
PROGRAM MULTIPLY
INTEGER(KIND=8)  IA(1:64),  IB(1:64),  IC(1:64),  DUMMY(1:64)

IA = ISHFT( -1_8, (/ ( I, I = 0,-63,-1 ) /) )          !See remark
IB = ISHFT( -1_8, (/ ( I, I = 0,-63,-1 ) /) )

PRINT *, '  Initial data:'
PRINT '( B64 )', IA
PRINT '( B64 )', IB

DUMMY(1:64)         = M@LD( IA (1:64))    !  Load the bit matrix IA transposed
IC(1:64)            = M@MX(IB(1:64))      !  Multiply IB with transposed IA
DUMMY(1:64)         = M@UL( )             !  Unload the unit

PRINT *, '  After multiplication:'
PRINT '( B64 )', IC
END
```

Remark

- This program runs at 1.6 Tbop/s!
- The integer values must be declared as KIND=8 to make them 64 bits wide.
- -1_8 should be understood as the integer value –1 in 8 bytes, which results in 64 bits set!

Appendix B

Some F90 and F77 implementation differences
The following section describes miscellaneous differences between using the F77 compiling system and the F90 compiler. For the true diehards F77 is still available: **fort77.**

- The F77 compiling system allows assumed-size character dummy procedures. The F90 compiler does not allow this.
- Because of the Fortran 90 rules of type conformance, the F90 compiler might be more restrictive than the F77 compiling system with regard to intrinsic assignment.
- The F77 compiling system allowed COMPLEX*4. No other vendor appears to allow this. The F90 compiler does not allow it, but it allows COMPLEX(KIND=4), which is equivalent to COMPLEX*8.
- The F90 compiler requires the RECURSIVE keyword for recursive routines. The following code compiles without error with the F77 compiling system, but an error is generated when compiled with the F90 compiler. The F90 treatment of this code complies with the Fortran 90 standard, and it allows the compiler to diagnose accidental causes of recursion:

```
INTEGER FUNCTION I( )
I = I( ) + 1
```

- Unlike the F77 compiling system, the F90 compiler generates a compile-time error when an IMPLICIT statement follows a PARAMETER statement if the information on the IMPLICIT statement contradicts the type of the named constant, as in the following example:

```
PARAMETER(A=1)
IMPLICIT INTEGER(A-Z)
PRINT *, A
```

  The F90 treatment of this code is consistent with the Fortran 90 standard, and it avoid potential ambiguities.

- Treatment typeless constants is different between the F77 compiling system and the F90 compiler in DATA statements. Consider the following code fragment

```
COMPLEX C1, C2
REAL R1, R2
DATA C1 /O'77'/
DATA C2 /Z'10'/
DATA R1 /X'77'/
DATA R2 /77B/
PRINT *, C1, C2, R1, R2
```

  Result: (63.,0.), (16.,0.), 0.E+0, 0.E+0

  The F77 compiler produces no diagnostic messages at compile time for the preceding code. The F90 compiler generates a compile-time message for each DATA statement. The F90 treatment of this code is consistent with the Fortran 90 standard and is more consistent than the F77 compiling system in handling constants.

- When the value of an expression depends on the order of evaluation, and the order of evaluation is processor dependent, the F90 compiler and the F77 compiling system may evaluate the items in a different order. This can lead to differences in the generated output. The following example code contains such expressions:

```
RBIG = 1.2E+83
R20 = RBIG - 1.2E+83 + 20          ! Use of variable in expression
PRINT *, "Expected: 20"
PRINT *, "Received: ", R20         ! Use of all constants in expression
R20 = 1.2E+83 - 1.2E+83 + 20
PRINT *, "Expected: 20"
PRINT *, "Received: ", R20
```

  The expression can yield different answers (e.g. 0.E0 or 20.) depending on whether the - or + operation is evaluated first. This is due to the large difference in magnitude of the operands and the fixed precision of the machine. The order of evaluation in these expressions is processor dependent, according to the Fortran 90 standard. The F90 compiler evaluates the operators of the first expression in a different order from the F77 compiling system.

- FORTRAN 77 provided one precision of integer, complex and logical data and two precisions of real data. Fortran 90 allows an implementation to have any number of precisions for these data types. Many vendors provided additional precisions as an extension to their FORTRAN 77 implementations through the type*byte_count form of declaration. The F77 compiling system accepted these extensions, but it mapped them onto the basic types required by the FORTRAN 77 standard. The F90 compiler, like the F77 compiling system, accepts this syntax but treats these additional data types as distinct types. This is done to allow for unambiguous resolution of procedure interfaces, overloaded operators, and user defined generics. Because of this difference between the F77 compiling system and the F90 compiler, some F77 programs that employ the type*byte_count syntax may not be accepted by the F90 compiler. These differences occur in the definition and use of statement functions. For example:

```
REAL*4 X
STMT_FUNC(R) = R + 1.0    ! Statement function
X = STMT_FUNC(X)
```

  The preceding program fragment would compile without error with the F77 compiling system. On a platform where the default real kind is 8, the F90 compiler issues an error for the statement function use because the statement function is defined with a default real argument but is passed a nondefault real actual argument. You can use the **–s** option on the **f90** command line to avoid receiving this error message.

- The F77 compiling system and the F90 compiler differ in the way overindexed code is handled. The F77 compiling system permitted constructs such as the following:

```
SUBROUTINE (A, N)
DIMENSION A(20, 400)
DO I = 1,N
        A(I) = . . .
END DO
```

The F90 compiler does not overindex if the leading dimension is known. Use the **–O overindex** option on the **f90** command if you want to overindex an array. For more information on f90, see **man f90**.