

The RWTH SunFire SMP-Cluster

User's Guide, Version 3.0

September 2002

**Dieter an Mey, Center for Computing and Communication, Aachen University
(Rechen- und Kommunikationszentrum der RWTH Aachen)**

anmey@rz.rwth-aachen.de

Ruud van der Pas, Application Performance Specialist, Sun Microsystems

ruud.vanderpas@sun.com

Eugene Loh, High-End Software, Sun Microsystems

eugene.loh@sun.com

Table of Contents

1 Introduction	4
2 Hardware	5
2.1 Configuration RWTH	5
2.2 Processors	6
2.3 Memory	7
2.4 Network RWTH	8
3 Operating System	9
3.1 Addressing Modes	9
3.2 Batch Job Administration RWTH	10
3.3 Defaults of the User Environment RWTH	13
3.4 User File Management RWTH	14
4 Programming/Tuning	15
4.1 Sun Compilers	15
4.2 The KCC C++ Compiler by KAI RWTH	19
4.3 Interval Arithmetic	19
4.4 Tuning Tips	19
4.5 Time measurements	21
4.6 Hardware Performance Counters	21
5 Parallelization	23
5.1 Message passing with MPI	23
5.1.1 Sun MPI	23
5.1.1.1 Placing the MPI-Tasks with mprun	23
5.1.1.2 Input and output control with mprun	24
5.1.1.3 Handling MPI program runs	25
5.1.1.4 Sun MPI environment variables	25
5.1.2 mpich RWTH	26
5.2 Shared memory programming with OpenMP	27
5.2.1 Sun-OpenMP	28
5.2.2 KAP/Pro Toolset RWTH	28
5.2.3 Automatic shared memory parallelization of loops	29
5.4 Hybrid Parallelization	30
6 Debugging	31
6.1 Static program analysis	31
6.2 Dynamic program analysis	31
6.3 Debuggers	32
6.3.1 dbx	32
6.3.2 Prism	34
6.3.3 TotalView	34
6.3.3.1 Invocation of TotalView for serial programs	34
6.3.3.2 Debugging of Sun-MPI programs RWTH	34
6.3.3.3 Debugging of OpenMP-programs	35
7 Programming tools	36
7.1 Sampling Collector and Performance Analyzer	36
7.1.1 The Collector	36
7.1.2 The Performance Analyzer	37
7.1.3 The Performance Tools Collector Library API	38

7.2 Frequency analysis with tcov	38
7.3 Run time analysis with gprof	39
7.4 Run time analysis of MPI programs	39
7.4.1 Sampling Collector and Performance Analyzer	39
7.4.2 Prism environment	39
7.4.3 Vampir and VampirTrace RWTH	41
7.4.4 Jumpshot and the MPE Library	41
8 Application software	43
8.1 Application software and program libraries RWTH	43
8.2 The Sun Performance Library	43
8.3 The Sun S3L library	43
8.4 Nag Numerical Libraries RWTH	43
9 Further information	45
9.1 Sun products	45
9.1.1 on Sun's web site	45
9.1.2 on the RWTH Support Server	45
9.1.3 on local file systems	45
9.2 Third party products	45
9.3 Public domain software	46
9.4 Problems and inquiries	46
10 Miscellaneous	47
10.1 Other Useful commands	47

1 Introduction

This primer gives you a quick start in using the new Sun Fire SMP-Cluster at the Aachen University.

It describes the hardware architecture, selected aspects of the operating environment, a few software tools, and helpful references for further information. The software tools include:

- Sun ONE Studio 7, Compiler Collection, formerly called Forte Developer Version 7, Sun's latest compiler and programming environment (It will be the default compiler from Sept/Oct 2002 on [RWTH](#))
- Sun HPC ClusterTools Version 4.0, Sun's MPI implementation and environment
- TotalView Version 5.0, Etnus' parallel debugger
- VampirTrace Version 2.0 and Vampir Version 2.5, Pallas' tools for runtime analysis of MPI programs
- KAP/Pro Toolset Version 4.0, KAI's OpenMP tools including
- the KCC C++ Compiler Version 4.0
- some details about the Solaris 9 operating system (It will successively be installed on the Sun Fire systems from 4Q 2002 on and particularly effects the runtime behaviour of the Sun Fire 15 K systems [RWTH](#))

Site-specific sections are marked with [RWTH](#).

Please check our web pages for more up-to-date information and the latest version of this document:

<http://www.rz.rwth-aachen.de/hpc/sun>

For interactive access to the cluster, login to

cluster-sun.rz.RWTH-Aachen.DE

For acquiring access to the machines, fill out the registration form at

<http://rz.rwth-aachen.de/hpc/org/antraege.html>

Please join the **rzcluster** mailing list, if you want to be informed on a regular basis:

<http://MailMan.RWTH-Aachen.DE/mailman/listinfo/rzcluster>

Do not hesitate to send criticisms or suggestions to

hpc@rz.rwth-aachen.de

Have fun using the new Sun Fire SMP-Cluster!

2 Hardware

2.1 Configuration RWTH

The Sun SMP cluster currently consists of

16 Sun Fire 6800 nodes with 24 UltraSPARC-III Cu processors and 24 GB of shared memory each and of

4 Sun Fire 15K nodes with 72 UltraSPARC-III Cu processors and 144 GB of shared memory each.

All 672 CPUs have a 900 MHz clock cycles with an accumulated peak performance of 1,2 TFlop/s and a total main memory capacity of 906 GB.

All compute nodes are equipped with local scratch (**TMP**) and system file systems. They also have access to a common NFS file system for small long-term user data (**HOME**) and to another common file system for large medium-term work files (**WORK**).

In the future (4Q2002) all compute nodes will have direct access to all shared file-systems via a fast storage area network (SAN) using the QFS file system. High IO bandwidth will be achieved by striping multiple RAID systems.

The SMP compute nodes are connected to each other by Gigabit Ethernet. In 1Q2003 proprietary high-speed Fire Link networks will be installed to form three clusters: two clusters of 8 Sun Fire 6800 systems each and one cluster of 4 Sun Fire 15K systems.

Finally all nodes will be upgraded with UltraSPARC-IV processors in 2H2003.

date	nodes	processors	main memory	networks
2Q2001	8 Sun Fire 6800	8 x 24 US-III 750	8 x 24 GB	GE, Myr
4Q2001	8 Sun Fire 6800 8 Sun Fire 6800	8 x 24 US-III 750 8 x 24 US-III 900	8 x 24 GB 8 x 24 GB	GE, Myr GE
1H2002	8 Sun Fire 6800 8 Sun Fire 6800 4 Sun Fire 15K	8 x 24 US-III 900 8 x 24 US-III 900 4 x 72 US-III 900	8 x 24 GB 8 x 24 GB 4 x 144 GB	GE GE GE
2H2003	8 Sun Fire 6800 8 Sun Fire 6800 4 Sun Fire 15K	8 x 24 US-IV 1500 8 x 24 US-IV 1500 4 x 72 US-IV 1500	8 x 24 GB 8 x 24 GB 4 x 144 GB	GE, Fire Link GE, Fire Link GE, Fire Link

2.2 Processors

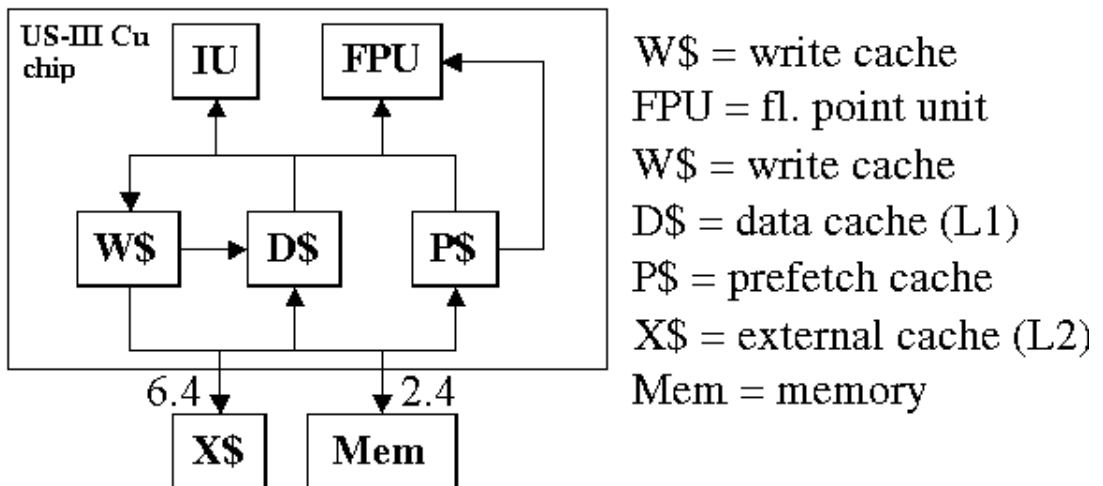
The UltraSPARC-III Cu processor (US-III Cu) is a superscalar 64-bit processor with two cache levels:

Level 1 (on chip):

- 64 KB for data and 32 KB for instructions
(4-way associative, 32 byte cache-lines, write-through, no-write allocate, pseudo random replacement strategy, 2 clock cycles latency. Modified cache lines are written back immediately into the L2 cache and a cache line is not fetched before a write operation)
- 2 KB prefetch cache, for an accelerated load of floating point numbers
(4-way associative, 64 byte cache lines, 32 byte subblocks, LRU replacement strategy)
- 2 KB write cache
(4-way associative, 64 byte cache lines, 32 byte subblocks, LRU replacement strategy)

Level 2 (off chip):

- 8 MB for data and instructions
(2-way associative (900 MHz), 512 Byte cache lines with 64 byte subblocks, approx. 12 clock cycles latency, 6.4 GB/s bandwidth, write-back; write-allocate strategy . Modified cache lines are not written back until they are pushed out of the cache and before a write the whole subblock has to be fetched from memory.)



The most important information about the current processors can be acquired with the instruction

\$ fpversion

Each clock period the processor can initiate 2 integer operations or an integer and a memory operation, one floating point addition and one floating point multiplication. Thus the peak performance in Mflop/s is twice the clock rate in MHz. In suitable computing kernels, like the well-known Linpack benchmark or a matrix multiplication, 70-90% of this rate will be actually attainable.

2.3 Memory

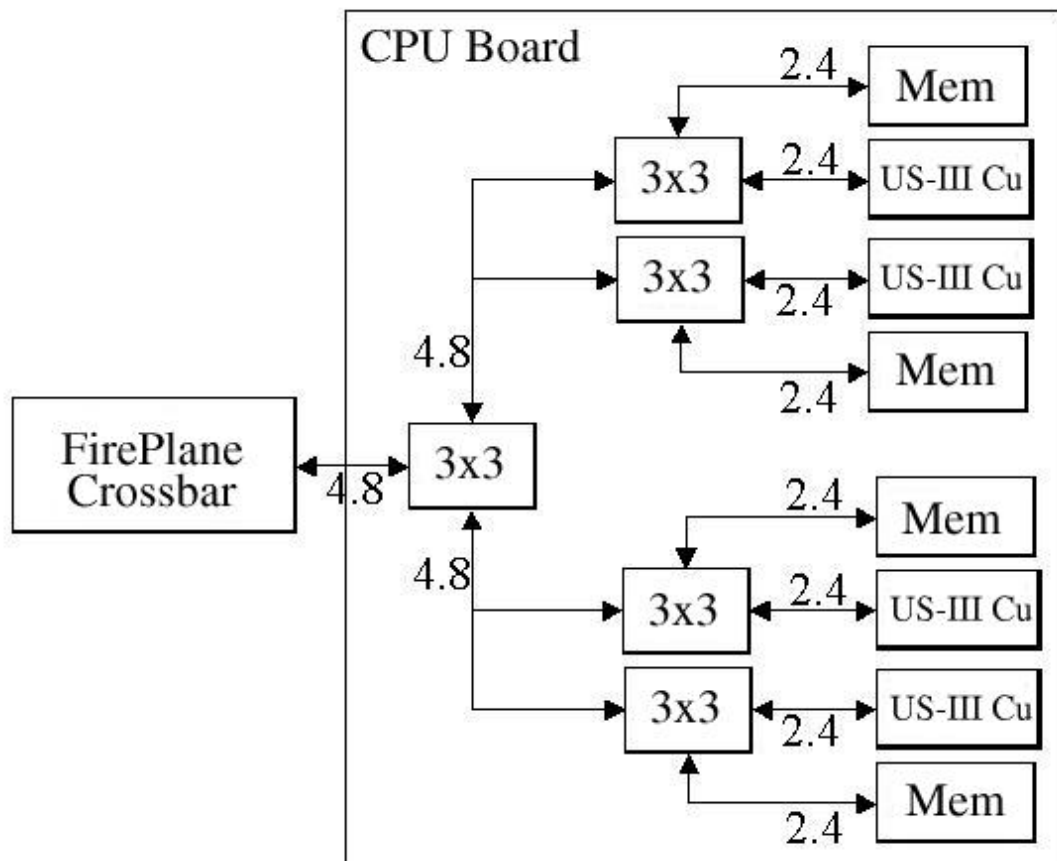
Each CPU board contains 4 processors and their external (L2) caches together with their local interleaved memory.

In the Sun Fire 6800, 6 of these boards are coupled with a crossbar. The memory bandwidth amounts to theoretically 2.4 GB/s for a single processor and -- due to snoop bus limitations -- 9.6 GB/s for all 24 processors of a SMP node.

In the Sun Fire 15K, 18 CPU boards are interconnected with a crossbar and the cache coherency is handled by a combination of snooping within each board and directory-based cache coherency between the boards.

From the programmer's point of view, the Fire 6800 thereby offers a "flat" memory system with a limited bandwidth (9.6 GB/s), i.e. all memory cells approximately have the same distance to each processor (latency about 270 ns), whereas data locality plays a more important role in the Fire 15K (cc-NUMA architecture).

The latency to get data from memory on the same board is approximately 270 ns. The Fire 15K's latency for fetching remote data will be at least 330 ns and in extreme cases, however, up to about 600 ns. Theoretically the total memory bandwidth will be between 43.2 GB/s (worst case) and 172.8 GB/s (only local accesses). Data locality will be supported by the upcoming version of the Solaris 9 operating system.



2.4 Network **RWTH**

Gigabit Ethernet will be used to interconnect SMP nodes. Further, two tightly coupled clusters of 8 Fire 6800 systems each and one cluster of 4 Fire 15K systems will be formed with a proprietary high-speed Fire Link network in 1H2003. With Sun's version of MPI, a latency of 4 micro seconds and a bandwidth of up to 2.7 GB/s can be obtained between nodes.

3 Operating System

The Solaris Operating Environment is an operating system of the UNIX family.

The current version on all Sun Fire machines is Solaris8. In the 2002/2003 time frame we will migrate to Solaris 9. **RWTH**

The command

```
$ uname -r
```

will print out the corresponding SunOS release level 5.8 or 5.9.

NEW: Solaris 9 will introduce the multiple page size support and the memory placement option (MPO) for the Sun Fire 15K systems.

By default Solaris organizes all data in pages of 8 KB. Programs with a large memory requirement and/or programs which access memory randomly or with non-unit strides might profit from using a large page size (reduction of TLB misses). This can be done by using the **ppgsz** command. Example:

```
$ man ppgsz # manual page
```

```
$ ppgsz -o heap=4M,stack=4M a.out # program start
```

All dynamically or locally allocated data, as well as uninitialized Fortran COMMON Blocks will be allocated on large pages (4 MB in the above example), if enough consecutive memory is available. You may check the running program with the command

```
$ pmap -s pid # address space map of a process
```

The Sun Fire 15K machines have a non uniform memory access (cc-NUMA) architecture. Thus processors have a quicker access to data on a memory chip on the same CPU board than to that residing on a different board. Whereas Solaris 8 allocates data uniformly on all boards, MPO in Solaris 9 tries to place data (pages) on the same board as the processor, which touches the data first (first touch policy)

3.1 Addressing Modes

Solaris 7, 8 and 9 are 64-bit UNIX operating systems. Programs can be compiled and linked in 32-bit mode (default) or 64-bit mode. This affects memory addressing (usage of 32- or 64-bit pointers) and has no influence on the precision of floating point numbers (4- or 8-byte real numbers). Programs needing more than 4 GB memory, have to use the 64-bit addressing mode. The switches for UltraSPARC-III Cu specific compilation and linking are

-xarch=v8plusb	32-bit
-xarch=v9b	64-bit

Note: **long int** data and pointers in C programs are stored with 8 bytes when using 64-bit addressing mode.

3.2 Batch Job Administration RWTH

Batchjobs are handled by the **Sun GridEngine** (formerly **Codine**).

Job scripts can be submitted to the batch system with the line command

```
$ qsub [options] [scriptfile | - [script_args] ]
```

or through the graphical user interface

```
$ qmon
```

The attributes of queued jobs can be modified with

```
qalter [options]
```

Jobs can be deleted with

```
$ qdel job_id
```

Status information can be inquired with

```
$ qstat -f | -j job_id | -u user
```

On overview of the current batch job load of the entire Sun Fire cluster can be obtained with the utility

```
jobinfo RWTH
```

The most important **parameters of qsub** are:

-o [hostname:]path	standard output file
-e [hostname:]path	standard error file
-j y n	merge error outputs into standard output
-l resource=value,...	specification of the necessary resources (see below)
-N name	job name
-pe parallel_environment ntask	processor count for the MPI environment (see below)
-v variable[=value]	set environment variables
-w v	only check the job parameters, do not submit (this does currently not work in combination with the -pe parameter)
-r n	no restart, in case of a system crash
-hold_jid job_id,...	start after the termination of the indicated job
-M mail_address	notification mail address
-m b e a s n	send notification mail at job begin end abort suspend send no mail

The most important **resource parameters** are (see **qsub**-parameter **-l**):

-l hostname=hostname	computer name. Normally the use of this parameter is not recommended.
-l h_rt=hh:mm:ss	required real time [[hours:]minutes:]seconds Default: 0:10:00 Maximum: 24:00:00
-l h_vmem=xxxX	virtual memory specification in bytes, KB, MB oder GB e.g. vmem=10M default: vmem=1M
-l num_proc=nthread	in case of shared memory parallelization: specification of the number of threads
-l ostype=sunos	start on the SunFire compute nodes.
-l solaris8 -l solaris9	during the migration period from Solaris 8 to Solaris 9 these resource parameters may be used. Normally the use of these parameters is not recommended.
-l march=sf-15k -l march=sf-6800k	jobs can be directed to the Sun Fire 15K or to the Sun Fire 6800 nodes. Normally the use of these parameters is not recommended.
-l software=#_of_licenses	the need for software licenses has to be specified. Currently the available licensed packages are: abaqus, ansys, cfx4, cfx5, gamess, gaussian, g98, linda, marc, matlab, tascflow The number of licenses normally equals 1.

MPI-Jobs have to be submitted into one of the following "**parallel environments**".
The number of MPI processes has to be specified (**nproc**)

-pe mpi_sunos_* nproc	the MPI job will be started on any of the Sun Fire machines
-pe mpi_sunos_6800 nproc	the MPI job will be started on Sun Fire 6800 nodes only
-pe mpi_sunos_15k nproc	the MPI job will be started on Sun Fire 15K nodes only
-pe mpi_sunos_1host nproc	all processes of the MPI job will be started on any one Sun Fire node. All MPI communication will use the common main memory.

The parameters can also be indicated as comment lines, starting with the characters "#\$", in the beginning of the job scripts. Command line parameters have higher precedence than the imbedded script flags.

Submitting a serial job:

```
$ qsub -o $HOME/aus.txt -j y -l ostype=sunos \  
-l h_rt=00:15:00 -l h_vmem=500M scriptfile
```

This corresponds to

```
#!/usr/bin/ksh  
#$ -o $HOME/aus.txt  
#$ -j y  
#$ -l h_rt=00:15:00  
#$ -l h_vmem=500M  
#$ -l ostype=sunos  
  
cd workdir  
program
```

Example of a batch job script for starting a Sun MPI program:

(The environment variable **MPRUN_FLAGS** is predefined by the batch system in order to direct the MPI processes to the reserved machines. The limits are per process limits, so in total 5 times 500 MB will be reserved.)

```
#!/usr/bin/ksh  
#$ -N MPI-Test-Job  
#$ -l h_rt=00:15:00  
#$ -l h_vmem=500M  
#$ -pe mpi_sunos_* 5  
#$ -l ostype=sunos  
cd workdir  
mprun program
```

Example of a batch job script starting an OpenMP or an autoparallel program:

(The environment variable **OMP_NUM_THREADS** is predefined by the batch system in order to start as many threads as processors have been reserved. 500 MB will be reserved on one node for all 5 threads.)

```
#!/usr/bin/ksh  
#$ -N OpenMP-Test-Job  
#$ -l h_rt=00:15:00  
#$ -l h_vmem=500M  
#$ -l num_proc=6  
#$ -l ostype=sunos  
cd workdir  
program
```

Hybrid Programs use a combination of MPI and OpenMP, where each MPI process is multi-threaded. Example of a batch job script starting a hybrid program:

(The environment variables **MPRUN_FLAGS** and **OMP_NUM_THREADS** are

predefined by the batch system. In this example 5 groups of 4 CPUs will be reserved.)

```
#!/usr/bin/ksh
#$ -N Hybrid-Test-Job
#$ -l h_rt=00:15:00
#$ -l h_vmem=500M
#$ -l num_proc=4
#$ -pe mpi_sunos_15k 5
#$ -l ostype=sunos
cd workdir
mprun program
```

3.3 Defaults of the User Environment **RWTH**

The login shell is the korn shell (**ksh**). It's prompt is symbolized by the dollar sign. Accordingly numerous initialization scripts follow this syntax. They must be started with

```
$ . scriptfile
```

Environment variables are set with

```
$ export variable=value
```

This corresponds to the C shell command

```
% setenv variable value
```

If you prefer to use a different shell, start any necessary initialization scripts before you change to your preferred shell.

```
$ . init_script
$ exec csh
%
```

or insert the appropriate commands into the **.profile** file in your **HOME** directory. The Korn shell prompt is indicated with a dollar sign, and the C shell prompt is indicated with an “percentage” symbol.

For the use of the Sun ONE Studio Compiler Collection environment and HPC ClusterTools, the environment variables **PATH** and **MANPATH** are already adapted in the user profile:

```
$ export PATH=${PATH}:\
/opt/SUNWspro/bin:/optSUNWhpc/bin
$ export MANPATH=${MANPATH}:\
/opt/SUNWspro/man:/optSUNWhpc/man
```

3.4 User File Management **RWTH**

Every user owns directories on shared file systems for small, long-term user files (**\$HOME=/home/username**) and for large, medium-term workfiles (**\$WORK=/work/username**). The **\$HOME** data will be saved regularly.

A directory for local scratch files (**\$TMP=/tmp/username/login_pid**) is accessible only on the respective node and will be automatically created before and deleted after the terminal session or the batch job

4 Programming/Tuning

4.1 Sun Compilers

Before long we will switch production from the current Forte Developer 6 Update 2 Compiler suite (Sun Fortran, C and C++ compilers) to the new **Sun ONE Studio 7 Compiler Collection**. You can choose the compiler version by modification of the search path with one of the following commands **RWTH**

```
$ . Forte61.init
$ . Forte62.init
$ . Forte70.init
```

We recommend that you always recompile your code with the latest compiler for performance reasons and bug fixes.

Check the compiler version which you are currently using with the option

```
-v
```

or with the command

```
$ dumpstabs object_file
```

Online information in addition to the manual pages can be found by directing your browser to the local file

```
file:///opt/SUNWspro/docs/index.html
```

or to the website

```
http://docs.sun.com
```

The compilers are invoked with the commands

```
$ cc, c89, f90, f95, CC
```

The appropriate manual pages are available. You can get an overview of the available compiler flags with the option

```
-flags
```

It is in general recommended to use the same flags for compiling and for linking.

The latest compiler release (7.0) does no longer contain a separate FORTRAN77 Compiler. Instead the Fortran95 compiler has a new flag, which might be used to compile legacy Fortran codes.

```
f95 -f77 -ftrap=%none
```

In general we do not recommend to switch off all error trapping (`-ftrap=%none`). If your program compiles and runs fine with the above compatibility flag, we suggest you repeat this without the `-ftrap=%none` option.

For further information check the f77(1) manual page.

Compute intensive program parts can be translated and linked with the optimization options (US-III Cu)

```
-fast -xarch=v9b (64 bit addressing) or  
-fast -xarch=v8plusb (32 bit addressing)
```

-fast is a macro expanding to several individual options, which are meant to give you the best performance with one single compile and link (!) option. Note however that the expansion of **-fast** might be different across the various compilers and can change between different compiler releases.

At present (version 7.0) **-fast** with the Fortran 95 compiler corresponds to the following list (see *manual page*):

```
-O5 -xarch=native -pad=local -xvector=yes  
-xprefetch=yes -xprefetch_level=2 -f -fsimple=2  
-fns -ftrap=common -libmil -dalign -xlibmopt  
-depend
```

with the C compiler:

```
-dalign -fns -fsimple=2 -ftrap=%none  
-xalias_level=basic -xarch -xbuiltin=%all -xdepend  
-xlibmil -xmemalign=8s -xprefetch=auto,explicit  
-xO5 -xtarget=native
```

and with the C++ compiler:

```
-dalign -fns -fsimple=2 -ftrap=%none -nofstore  
-xarch -xlibmil -xlibmopt -xmemalign -xO5  
-xtarget=native -xbuiltin=%all
```

For further optimization by the C-compiler the following options can be added:

```
-xvector -xspfconst
```

and for further optimization by the C++-compiler the following options can be added:

```
-xalias_level -xvector -xspfconst  
-xprefetch=auto,explicit
```

The generated code can be specifically tuned for the 900 Mhz-UltraSPARC-III Cu processor (US-III Cu) by specifying

```
-xchip=ultra3cu -xcache=64/32/4:8192/512/2 \  
-xarch=v8plusb (32-bit addressing mode)  
-xchip=ultra3cu -xcache=64/32/4:8192/512/2 \  
-xarch=v9b (64-bit addressing mode)
```

(Use **-xchip=ultra** or **-xchip=ultraplus** with the version 6.1 resp. 6.2 compilers)

In general it is recommended to specify the precise architecture flags for linkage as well (**-xarch=v8plusb / v9b** for the UltraSPARC-III Cu processor), so that the optimal run time libraries are used.

You can get a survey of the compiler flags used by adding the option

```
-v (Fortran and C++)  
-# (C)
```

The compiler supports inlining of function and subroutine calls. With optimization level **-xO4** and above, this is attempted for functions/subroutines within the same source file. The programmer can also specify which functions/subroutines should be

inlined, by specifying these with the following option

```
-xinline=routine_list
```

Note however that in this case, automatic inlining is disabled. It can be restored through the %auto option. We therefore recommend the following:

```
-xinline=%auto,routine_list
```

If one wishes to have the compiler perform inlining across various source files, the -xipo option can be used. This is a compile and link option. With the 7.0 release, -xipo=2 is also supported. This adds memory related optimizations to the interprocedural analysis.

Program kernels with numerous branches can be further optimized with the profile feedback method. This two step method starts with a compile using this option added to the regular optimization options:

```
-xprofile=collect:a.out
```

Then the program should be run for one or more data sets. During these runs, run-time characteristics will be gathered.

The second phase consists of a re-compile, using the run-time statistics:

```
-xprofile=use:a.out
```

This will then hopefully give a better optimized executable, but keep in mind this is of benefit for specific scenario's only.

NOTE : High optimization can have an influence on floating points results due to different **rounding errors**. In order not to change the order of the arithmetic operations by the optimization, a further option can be added, which reduces the execution speed however:

```
-fast -fsimple=0 -xnolibmopt (Fortran)
```

The option

```
-g
```

produces **debugging information**. This is also useful for run-time analysis with the Performance Analyzer, which can use the debugging information to attribute time spent to particular lines of the source code. Use of **-g** does not substantially impact optimizations performed by the new Sun compilers. Meanwhile, the correspondence between the binary program and the source code is weakened by optimization, making debugging more difficult.

The Fortran compiler prints a lot of information (compiler messages, cross reference list, etc.) about the program in a separate **listing file** when compiling with the option

```
$ f90 -Xlist ... program.f  
$ cat program.lst
```

The **default data mappings** of the Fortran compiler can be adjusted with the **-typemap** option. The normal setting is

```
-typemap=real:32,double:64,integer:32 ...
```

For example with

```
$ f90 -typemap=real:64,double:64,integer:32 ...
```

the REAL type can be mapped to 8 bytes.

When using the `-g` option, the latest Sun compilers introduce comments about loop optimizations into the object files, which can be output by the command

```
$ er_src progname.o
```

A comment like

```
Loop below pipelined with steady-state cycle  
count..
```

indicates that modulo scheduling (aka software pipelining) has been applied, which in general gives better performance.

An expert of the chip architecture will be able to judge by the additional information, if further optimizations are possible.

With the help of the `er_src` command a successful subroutine inlining can also be easily verified:

```
er_src *.o | grep inline
```

NOTE: The compiler options are interpreted from left to the right. In the case of contradictory options the right most dominates.

4.2 The KCC C++ Compiler by KAI RWTH

KCC is an excellent C++ compiler by Kuck & Associates. KCC translates C++ programs to an intermediate C code, which then can be compiled by a native C compiler. KCC is imbedded in the **guidec++** OpenMP compiler.

The most important KCC flags are

+K3	maximum optimization
-O<n> resp. -fast	optimization level of the back-end C compiler (will be passed through)
-k or -keep_gen_c	do not delete the generated intermediate C code. The C code stored into <filename>.int.c might be interesting, but it is hard to read.
-v	verbose mode
--backend ...	pass the following option to the back end compiler
-c, -o	will be passed through as well.

4.3 Interval Arithmetic

The Sun Fortran and C++ compilers support interval arithmetic by an intrinsic **INTERVAL** data type and the UltraSPARC-III Cu processor supports fast switching of the rounding mode of floating point operations.

The use of interval arithmetic requires the use of appropriate numeric procedures.

4.4 Tuning Tips

Compiler options, compiler directives, programming techniques and last but not least the Sun performance library with highly optimized routines can be used for accelerating programs.

Recently an excellent book covering this topic particularly on UltraSPARC computers has been published:

[Rajat Garg and Ilya Sharapov Techniques for Optimizing Applications: High Performance Computing, ISBN:0-13-093476-3, published by Prentice-Hall PTR/Sun Press.](#)

Contiguous memory access is critical for reducing cache and TLB misses. This has a direct impact on the addressing of multidimensional fields or structures. Therefore Fortran arrays should be processed in columns and C and C++ arrays in rows. When using structures, all structure components should be processed in quick succession. Frequently this can be achieved by the technique of the *loop interchange*.

The limited memory bandwidth of RISC processors like the UltraSPARC III is a severe bottleneck for many scientific applications. With *prefetching* data can be loaded in advance from the memory into the cache and into the registers. This can be supported automatically by hard- and software but also by explicitly adding prefetch directives resp. calls.

The re-use of cache contents is very important, in order to reduce the number of memory accesses. If possible block algorithms should be used e.g. from the optimized Sun performance library described below.

Cache behavior of programs can be improved frequently by the techniques of *loop*

fission (=loop splitting), by *loop fusion* (=loop collapsing), by *loop unrolling* (see option **xunroll=n**), by loop blocking, the strip mining and by combinations of these methods. Conflicts caused by the mapping of storage addresses to cache addresses can be eased by the creation of buffer areas (*padding*) (see compiler option **-pad**).

With the option **-dalign** the memory access on 64 bit data can be accelerated. This alignment permits the compiler to use single 64 bit load and store instructions. Otherwise, the program must use more than one instruction for each memory access. However this option must be applied to each routine.

With this option, the compiler will assume that double precision data has been aligned on an 8-byte boundary. If the application violates this rule, the run-time behaviour is undetermined, but typically the program will crash.

On well-behaved programs, this should not be an issue, but care should be taken for those applications that perform their own memory management, switching the interpretation of a chunk of memory while the program executes. A classical example can be found in some (older) Fortran programs. A large INTEGER COMMON -block is allocated, but later on this is declared to be a DOUBLE PRECISION COMMON -block of half the size. Under such circumstances, a misalignment of data can easily happen.

NOTE: The **-dalign** options is actually *required* for Fortran MPI programs and for programs linked to other libraries like the Sun Performance Library and the NAG libraries.

The compiler optimization can be improved by integrating frequently called small subroutines into the calling subroutines (*inlining*). The expense for the subroutine call will be avoided thereby.

-xinline=routine1, routine2, . . .

(*Inlining* of routines from the same source file)

-xO4 -xcrossfile

(*Inlining* of routines from other files in the same compiler call)

-xipo

(*Inlining* of routines from other files in different compiler calls)

In C and C++ programs the use of pointers frequently obstructs the possibility for optimization by the compiler. Through compiler options

-xrestrict and **-xalias_level= . . .**

it is possible to give additional information to the C-compiler.

With the directive

#pragma pipelooop(0)

in front of a **for** loop it can be indicated to the C-compiler that there is no data dependency present in the loop.

Word of caution. These options and the pragma make certain assumption. When using these mechanisms incorrectly, the behaviour of the program becomes undefined.

Please study the documentation carefully before using these options or directives.

4.5 Time measurements

For real time measurements a high-resolution timer is available. However, the measurements can supply reliable, reproducible results only on an (almost) empty machine. At least the number of runnable processes (use **uptime** command) plus the number of processors needed for the measurement has to be by far less than the number of processors available on the compute node.

Example in C

```
#include <sys/time.h>
/* Real time in nanoseconds as long long int */
double second;
second = (double) gethrtime() * 1.0E-9;
```

and in Fortran

```
INTEGER*8 gethrtime
REAL*8 second
second = 1.d-9 * gethrtime()
```

CPU time measurements have a smaller precision and are more time costly. For measuring large time intervals they are quite suitable.

In case of parallel programs, real time measurements should be made anyway!

After linking a library supplied by the computing center:

```
-L/usr/local_rwth/lib -lr_lib RWTH
```

the functions **r_rtime** and **r_ctime** are available. They return the real time and the CPU time, respectively, as double floating point numbers.

4.6 Hardware Performance Counters

The UltraSPARC-III Cu Chip offers 2 programmable 32-bit performance counters for counting various hardware events.

The **cputrack** command (see **man cputrack**), the **cpc-library** (see **man cpc**), the portable PCL-library or the Performance Analyzer (see chapter 7) can be used to access these counters.

The command

```
$ cputrack -h
```

lists the names of the countable events. A simple application can be seen in the shell script

```
/usr/local_rwth/bin/mflops RWTH
```

Just call

```
$ /usr/local_rwth/bin/mflops a.out RWTH
```

to count the number of floating points instructions during the execution of **a.out** in MFlop/s.

```
$ man cpc_bind_event
```

displays an example program using the cpc library.

The portable performance counter library (PCL) profits from the cpc library. It can be linked by

```
$ f90 -L/usr/local_rwth/lib -lpcl -lcpc ...
```

A more elegant way of obtaining performance information is the use of the **collect** command and the **er_print** utility or the **analyzer** GUI (see chapter 7)

5 Parallelization

Parallelization for computers with shared memory (SM) means either the automatic distribution of loop iterations on several processors or the explicit distribution of work on the processors by compiler directives and function calls (OpenMP) or a combination of both.

Parallelism for computers with distributed memory (DM) is done via the explicit distribution of work and data on the processors and their coordination with the exchange of messages (Message Passing with MPI).

MPI programs run on shared memory computers as well, whereas OpenMP programs (normally) do not run on computers with distributed memory. As a consequence MPI programs can use all available processors of the SMP cluster, whereas OpenMP programs can use up to 24 processors of a Sun Fire 6800 node, or up to 72 Processors of a Sun Fire 15K node.

For large applications the hybrid parallelization approach, a combination of coarse-grained parallelism with MPI and underlying fine-grained parallelism with OpenMP, might be an attractive possibility, in order to use as many processors efficiently as possible.

5.1 Message passing with MPI

5.1.1 Sun MPI

Sun MPI is the Sun implementation of the MPI standard and is part of the Sun HPC ClusterTools software suite. At present, ClusterTools 4.0 is installed.

The compiler drivers **mpf77**, **mpf90**, **mpcc** and **mpCC** and the instruction for starting an MPI application **mprun** are in the directory **/opt/SUNWhpc/bin**. The necessary include directory **/opt/SUNWhpc/include** and the library directory **/opt/SUNWhpc/lib** are picked up automatically by these compiler drivers.

Example (recommendation):

```
$ mpf90 -c -dalign ... *.f90
$ mpf90 -o a.out *.o -lmpi
$ mprun -np 4 a.out
```

Example (only for explanation):

```
$ f90 -I /opt/SUNWhpc/include -c -dalign ... *.f90
$ f90 -o a.out *.o -L/opt/SUNWhpc/lib -lmpi
$ /opt/SUNWhpc/bin/mprun -np 4 a.out
```

MPI programs can be started with the command

```
$ mprun [options] program
```

The command **mprun** has numerous flags for placing the MPI tasks on the compute nodes and for input and output control (see also **man mprun** and **mprun -h**).

5.1.1.1 Placing the MPI-Tasks with **mprun**

The following table contains the most important parameters of **mprun** for the distribution of the MPI tasks on the involved machines.

Please note however, that large computing jobs should not be started interactively,

and that with use of batch jobs (see chapter 3), the GridEngine batch system determines the distribution of the MPI tasks on the machines to a large extent.

Small MPI test jobs can be started on the interactive node, where you use to logged in by just specifying

```
mprun -np n program
```

because the environment variable MPRUN_FLAGS is predefined in the user profile such that all MPI processes will be started on the current machine.

-J	Prints the job identification number
-np n	Start of exactly n MPI tasks
-np 0	Start of exactly one MPI task for each processor
-S -np n	Start n MPI tasks, but settle for one process per CPU if not enough CPUs are available.
-W -np n	Cyclic distribution of the MPI tasks on the processors, if the number of MPI tasks is larger than the number of the processors of the SMP node.
-np n -l "sunc01.rz.RWTH-Aachen.DE 2, sunc02.rz.RWTH-Aachen.DE 3"	Explicit distributing of the <i>n</i> MPI tasks to the indicated SMP nodes. Note: capitalization is relevant
-np n -m rankmapfile	Explicit distribution of the <i>n</i> MPI tasks on SMP nodes listed in a file.
-np n -Ns	Start of exactly one MPI task on each of <i>n</i> SMP nodes.
-Zt m -np n	Start of <i>n</i> MPI tasks in groups of <i>m</i> on each of the involved nodes.
-Z m -np n	Start of <i>n</i> MPI tasks in groups of <i>m</i> . Several groups on a sufficiently large SMP node are allowed.

5.1.1.2 Input and output control with mprun

Under normal conditions standard input (**stdin**) is passed to all MPI tasks by the **mprun**, command. Standard output (**stdout**) as well as standard error output (**stderr**) of all tasks are passed to the standard output of **mprun**.

By further options of the command **mprun** this behavior can be modified:

-D	The error outputs of the tasks are passed to the error output of the mprun command.
-N	All standard input and output is turned off.
-n	/dev/null is passed to the standard input. That can be useful for MPI jobs, which run in the background (e.g. as a batch job), so that they do not block, if they wait unintentionally for an input. In this case they will read an EOF.
-B	The output of the tasks is written to files named out.jid.rank .
-o	The output is buffered line by line and the rank of the respective process is written on the beginning of each line.
-I „0r,1w1,2w1“	more precise controlling of the input and output. Only complete lines will be written.
-I „0r,1wt,2wt“	Only complete lines are output and all lines have the task rank placed in front.
-I „0r=input, \ 1wt=out.&J.&R, \ 2w=err.&J.&R“	All tasks read the same input file, but write in separate output and error output files

5.1.1.3 Handling MPI program runs

You can terminate a MPI job with the job identification number *jid* (see: **mprun -J**) by:

```
$ mckill jid
```

The command **mpps** gives a list of the processes, that run under the control of the MPI run time system (**CRE=cluster run time environment**).

```
$ mpps -pef
```

The command **mpinfo** gives an overview of the configuration of all nodes attached to the CRE. Example:

```
mpinfo -N
```

5.1.1.4 Sun MPI environment variables

Numerous environment variables can govern the behavior of an MPI program. Occasionally performance can be improved by increasing internal buffering with

```
$ export MPI_SHM_SBPOOLSIZE=8000000
$ export MPI_SHM_NUMPOSTBOX=256
```

Another environment variable should be set in combination to this. The value depends on the number of address bits. With 32 bit addressing, the command is

```
$ export MPI_SHM_CYCLESTART=0x7fffffff
```

and with 64 bit addressing

```
$ export MPI_SHM_CYCLESTART=0x7fffffffffffffff
```

should be set.

It is typically safe and helps performance when one decouples processes by suppressing "cyclic" message passing. This is done for 32-bit addressing with

```
$ export MPI_SHM_CYCLESTART=0x7fffffff
```

and for 64-bit addressing with

```
export MPI_SHM_CYCLESTART=0x7fffffffffffffff
```

In the case of exclusive use of the involved SMP nodes, in particular if one processor in each node is kept free for system processes, it's possible to accelerate a program with:

```
$ export MPI_SPIN=1
```

The MPI tasks wait then actively (busy waiting, spinning) for messages and keep their processor busy thereby.

In some cases (e.g. pingpong tests)

```
$ export MPI_POLLALL=0
```

accelerates the application (do not poll).

In case of problems more run time messages can be printed through

```
$ export MPI_SHOW_INTERFACES=3
```

```
$ export MPI_SHOW_ERRORS=1
```

```
$ export MPI_CHECK_ARGS=1
```

The current values of all MPI related environment variables will be listed at the program start with:

```
$ export MPI_PRINTENV=1
```

Appendix A of the [Sun HPC ClusterTools Performance Guide](#) sums up tips for the tuning of MPI applications.

5.1.2 mpich RWTH

Alternatively to the described use of the Sun MPI, which is optimised for the Sun computer architecture, also the public domain version **mpich 1.2.4** can be used (at present only with 32 bit addressing). The instruction

```
$ . mpich.init
```

initializes the environment.

Example (recommendation):

```
$ . mpich.init
```

```
$ mpif90 -o a.out ... *.f90 -lmpich
```

```
$ mpirun -np 4 a.out
```

Example (for explanation):

```

$ export MPICH_ROOT=/rwthfs/rz/SW/MPI/\
    mpich-1.2.4/solaris/ch_p4
$ f90 -I ${MPICH_ROOT}/include -c ... *.f90
$ f90 -o a.out *.o -L${MPICH_ROOT}/lib \
    -lmpich -lthread -lsocket -lnsl -laio
$ ${MPICH_ROOT}/mpirun -np 4 a.out

```

Note: The Sun MPI and mpich include files are not compatible. If mpif77 or mpif90 are aborted prematurely, then they leave links to **mpif.h** in the current directory, which might cause interesting errors, if one uses Sun MPI afterwards.

Usually Sun MPI should be preferred on the Sun Fire cluster.

5.2 Shared memory programming with OpenMP

For shared memory programming OpenMP is becoming the de facto standard. The OpenMP API is defined for FORTRAN, C and C++ and consists of compiler directives, run time routines and environment variables.

In the *parallel regions* of a program several *threads* are started, that execute the contained program segment redundantly, until they hit a *worksharing construct*. Within this construct, the contained work (usually **do-** or **for-**loops) is distributed among the threads. Under normal conditions all threads have access to all data (*shared* data). But pay attention: if data, accessed by several threads, is modified, then the access to this data must be protected in *critical regions*.

Also *private* data areas can be used, where the individual threads hold their temporary data. All local data of subroutines, which are called within parallel regions, are put on the stack, and thus don't keep their contents from one call to the next!

Therefore, Fortran programs must be translated with the option **-stackvar**. **COMMON** blocks, data in modules or **SAVE** statements must be used with caution (*thread safety*).

Attention! In many cases, the stack area for the slave threads must be increased by changing the environment variable **STACKSIZE**, or the stack area for the master thread must be increased with the (Korn shell) command **ulimit** (specification in KB). It is recommended to use the new compiler option (version 7.0)

```
-xcheck=stkovf
```

in order to detect stack overflow at runtime.

Hint: In a loop, which is to be parallelized, the results must not depend on the order of the loop iterations! Try to run the loop backwards in serial mode. The results should be the same. (This is a necessary, but not a sufficient condition!)

The number of the threads to use is indicated by the environment variable **OMP_NUM_THREADS**.

Notes: If **OMP_NUM_THREADS** is not set, then Sun OpenMP starts only 1 thread (as opposed to the Guide compiler from the KAP/Pro Toolset which starts as many threads as there are processors available).

On a loaded system fewer threads may be employed than specified by this environment variable, because the dynamic mode is used by default (as opposed to the

Guide compiler). Use the environment variable **OMP_DYNAMIC** to change this behaviour.

5.2.1 Sun-OpenMP

By adding the option

```
-xopenmp
```

the OpenMP directives (according to the latest OpenMP 2.0 specifications) are interpreted by the Fortran95 compiler. This option is an abbreviation for

```
-mp=openmp -explicitpar -stackvar -D_OPENMP -O3
```

Fortunately, the explicit parallelization can be combined with the automatic parallelization of the Fortran compiler. Loops within parallel OpenMP regions are no longer subject to automatic parallelization. Nested parallelization is not (yet) supported.

The C- and C++-compilers support OpenMP as well after adding the option

```
-xopenmp
```

Though the OpenMP features of the C++-compiler (version 7.0) are still quite limited.

Between parallel regions the slave threads go to sleep. How they are woken up is controlled by the environment variable **SUNW_MP_THR_IDLE**. The possible values are:

```
$ export SUNW_MP_THR_IDLE=sleep | ns | rns
```

The slave threads wait either actively (busy waiting, by default) and thereby consume CPU time or passively (idle waiting) and must then be woken up by the system or in a combination of these methods they wait first actively and fall asleep *n* seconds or *n* milliseconds later. With fine-grained parallelization active waiting and with coarse-grained parallelization passive waiting is recommended. Idle waiting might be advantageous on an overloaded system.

Setting

```
export SUNW_MP_WARN=TRUE
```

enables additional warning messages of the OpenMP run time system.

Use the new Fortran compiler option

```
-XlistMP
```

to receive additional OpenMP related messages in the listing files (***.lst**)

5.2.2 KAP/Pro Toolset [RWTH](#)

The KAP/Pro Toolset from the Kuck & Assoc. Inc. (KAI) contains OpenMP compilers and tools.

The Guide compilers interpret OpenMP directives in Fortran, C and C++ programs and generate intermediate programs with calls to the pthread library.

By just replacing the compiler and linker calls to

```
$ guidef77 | guidef90 | guidec | guidec++
```

appropriate compiler drivers are used.

By adding the

-WGkeep

flag the intermediate programs are kept. By linking with the option

-WGstats

a statistics file is written during program execution, which can be nicely visualized with

\$ guideview.

A remarkable tool for the verification of the correctness of OpenMP programs is **Assure**. Replacing the compiler and linker calls by

```
$ assuref77|assuref90|assurec|assurec++ \  
-WGpname=project
```

the program is instrumented, such that during the program execution every memory access is traced in order to detect possible data races.

The results of this analysis can be displayed with the GUI

```
$ assureview -pname=project
```

or printed out in line mode by

```
$ assureview -txt -pname project
```

The instrumented program will take about 10 times more CPU time and 10 times more memory!

Recommendation: Never put an OpenMP code into production, before using Assure!

5.2.3 Automatic shared memory parallelization of loops

The Sun Fortran- and C-compilers are able to parallelize loops automatically.

Success or failure to do so depends on the compiler's ability to be able to prove it is safe to parallelize a (nested) loop. This is often application area specific (e.g. finite differences versus finite elements), language dependent (pointers may make the analysis hard) and coding style.

The respective option is

-xautopar

The **-autopar** option is an abbreviation for

```
-xautopar -depend -xO3
```

The combination of explicit parallelism by directives and automatic parallelism is accessible by the option

-xparallel

as an abbreviation for

```
-xautopar -xexplicitpar -depend -xO3
```

Not only OpenMP directives are interpreted, but also proprietary parallelization directives of Sun and Cray, which, since OpenMP becomes more and more a standard, should not be used anymore. Adding

-mp=openmp

limits the compiler to OpenMP directives, if for historical reasons also different directives should be still contained in the program.

With the option

-xreduction

automatic parallelization of reductions is also permitted, e.g. accumulations, dot products etc., whereby the modification of the sequence of the arithmetic operation can cause different rounding error accumulations.

Compiling with the option

-xloopinfo

supplies information about the parallelization.

If the number of loop iterations is unknown during compile time, then code is produced, which decides at run-time whether a parallel execution of the loop is more efficient or not (*alternate coding*).

Also with automatic parallelization, the number of the used threads can be specified by the environment variable **OMP_NUM_THREADS**.

5.4 Hybrid Parallelization

The combination of MPI and OpenMP and/or autparallelization is called hybrid parallelization. Each MPI process is multi-threaded. It is important to link the thread-safe version of the MPI library:

```
$ mpf90 -openmp -fast -c program.f90
$ mpf90 -openmp -fast -o a.out program.o -lmpi_mt
$ export OMP_NUM_THREADS=n
$ mprun -np m a.out
```

KAI's guide preprocessors can be used as well:

```
$ guidef90 -Wgcompiler=mpf90 -openmp -fast -c \
  program.f90
$ guidef90 -Wgcompiler=mpf90 -openmp -fast \
  -o a.out program.o -lmpi_mt
$ export OMP_NUM_THREADS=n
$ mprun -np m a.out
```

6 Debugging

If your program is causing problems, it might be good opportunity to lean back and think for a while.

Take a step back:

What were the last changes that you made? (A source code revision system (RCS, CVS) might help.)

Reduce the number of CPUs in a parallel program, try a serial program run if possible.

Reduce the optimization level of your compilation,

Chose a smaller data set. Try to build a specific test case for your problem.

Look for compiler messages and warnings. Use tools for a static program analysis (see chapter 6.1)

Try a dynamic analysis with appropriate compiler options (see chapter 6.2). In case of an OpenMP program, use Assure (see chapter 5.2.2)

Use a debugger. Use the smallest case which shows the error. (see chapter 6.3)

6.1 Static program analysis

First, an exact static analysis of the program is recommended for error detection. The Fortran compiler offers an **-Xlist** option which generates warning and error messages into additional listing files (file extension **.lst**). For OpenMP programs there is a new option **-XlistMP**. Furthermore the following tools can be used for static analysis:

cc -v ..	stricter semantic checks of C programs by the compiler
lint	more accurate syntax check of C programs
ftnchek	more accurate syntax check of Fortran77 programs
foresys	more accurate syntax check of Fortran77 and Fortran90 programs and more

Sometimes, program errors occur only after high optimization by the compiler. That can be a compiler error or a programming error. If the program runs correctly without compiler optimizations, the module causing the trouble can be found by systematic bisectioning.

6.2 Dynamic program analysis

The program can be further checked by translation with certain options:

-C	array bound check of Fortran programs
-Xlist	global program analysis, write detailed list to files with the ending .lst
-ftrap=%all	pursue of floating point errors, like division by zero, overflow, underflow. The error handling can be programmed also explicitly, see: man ieee_handler
-g	enrich the binary program with debugger information, for step-by-step debugging, turn off all optimizations)
-xcheck=stkovf	check stack overflow at runtime, new with version 7

The sampling collector (see chapter 7.1) is now also able to detect memory leaks

```
collect -H
```

A core dump can be analyzed with the debugger, if the program was translated with **-g**:

```
$ dbx a.out core
$ totalview a.out core
```

If a program with optimization delivers other results than without, then the changed rounding error behavior can be responsible. There is a possibility to test this by optimizing the program “carefully”:

```
$ f90 ... -fsimple=0 -xnolibmopt...
```

Thus, the sequence of the floating point operations is not changed by the optimization, which can increase the run time.

6.3 Debuggers

At present four different debuggers are available. In all cases the program must be translated and linked with the option **-g** and without optimization (at least in the interesting program parts).

Don't forget to increase the core file size limit of your shell, if you want to analyze the core that your program may have left behind:

```
ulimit -c unlimited
```

But please don't forget to purge core files afterwards!

6.3.1 dbx

dbx is a powerful line orientated debugger with a detailed online help.

It can as well be used to debug long running programs in batch mode. Collect the **dbx** commands in an input file and start your program under control of **dbx**:

```

$ cat >> dbx.in < EOF
catch FPE
catch SIGSEGV
catch SIGBUS
run inputfile
where
dump
quit
EOF
$ dbx a.out < dbx.in

```

You may as well debug MPI-Programs this way:

```
mprun -np ntasks -o dbx a.out < dbx.in
```

It might be more comfortable only to run a few MPI processes through the debugger. This can be accomplished by starting a small shell script like the following:

```

#!/bin/ksh
# giving the corefile a useful name ...
coreadm -p core.%n.%f.p%p.j${MP_JOBID}.t$MP_RANK
$$;
# mechanism to restrict debugging to a subset of
MPI processes ...
if [[ $MP_RANK < 2 ]]
then
    dbx a.out < dbx.in > dbx.out.t$MP_RANK
    mpskill -9 $MP_JOBID
else
    debug.exe
fi

```

This script, using the same input file `dbx.in` for `dbx` like above, is than run with

```
mprun ... rundebbug.ksh
```

This will leave some core files with meaningful names behind, which then can be analyzed with

```
dbx a.out core.machinename.a.out.pnnnn.jmmmm.tkk
```

6.3.2 Prism

`prism` is a graphic debugger and run time analysis tool for Sun MPI programs.

If the help information browser does not start correctly, use

```
$ export PRISM_BROWSER_SCRIPT=yes
```

For debugging it can be helpful to turn off the following timeout setting:

```
$ export MPI_INIT_TIMEOUT=-1
```

Invoke `prism` by

```
$ prism -np n [-W] program
```

```

$ prism -np n [-W] -mprunargs `mprun_options`
program
$ prism program corefile
$ prism program jid

```

6.3.3 TotalView

The state-of-the-art debugger TotalView from Etnus (<http://www.etnus.com/>) can be used to debug serial and parallel Fortran, C and C++ programs. It is available on all major platforms.

At present **TotalView** does not yet support the latest Sun compiler version 6 Update 2. After

```
$ . Forte61.init RWTH
```

the 6 Update 1 compiler is in the search path.

The upcoming version 6 of TotalView will support the latest Sun compilers as well.

6.3.3.1 Invocation of TotalView for serial programs

```

$ . totalview.init
$ totalview program [corefile]

```

6.3.3.2 Debugging of Sun-MPI programs **RWTH**

At present a preliminary wrapper program has to be used, in order to debug a program compiled and linked with Sun MPI:

```

$ . totalview.init
$ totalview tvrun -a -np 2 -l "$(hostname) 2" a.out

```

Initially a dummy program is displayed in the source window. After pressing **Go** the user program is entered. A few preferences should be changed beforehand:

Add **/opt/SUNWhpc/lib** in

```
File - Preferences - Dynamic Libraries
```

and turn on **Stop the goup** in

```
File - Preferences - Parallel
```

Save these settings with **Save**.

Now start your parallel program with **Go**, the source of your main program will be displayed and the debugging session starts.

Message Queues cannot yet be displayed.

Programs using the public domain MPI library **mpich** can be debugged with TotalView as well:

```

$ . totalview.init
$ . mpich.init
$ ${MPICH_ROOT}/bin/mpirun -tv -np nproc program

```

6.3.3.3 Debugging of OpenMP-programs

Before debugging an OpenMP program, the corresponding serial program should

run correctly. The typical OpenMP parallelization errors are data races, which are hard to detect in a debugging session, because the timing behaviour of the program is heavily influenced by debugging.

It is recommended to use the new Fortran compiler option

-xlistMP

to do a basic static program check. Furthermore the **Assure** tool is recommended for the verification of OpenMP programs (see chapter 5.2.2).

But interactive debugging is possible as well. The Sun compilers' OpenMP options require high optimisation (-xO3) which in turn prohibits debugging. It is necessary to use KAI's guide precompiler, which can be combined with TotalView.

Example:

```
$ . Forte61.init
$ f90_6_1=$(which f90)
$ guidef90 -WGcompiler=$f90_6_1 \
  -WG,-cmpo=i -g -c *.f90
$ guidef90 -WGcompiler=$f90_6_1 \
  -WG,-cmpo=i -g -o a.out *.o
$ export OMP_NUM_THREADS=2
$ . totalview.init
$ totalview a.out
```

For the interpretation of the OpenMP directives, the original source program is transformed. The *parallel regions* are *outlined* into separate subroutines. *Shared* variables are passed as call parameters and *private* variables are defined locally. A parallel region cannot be entered stepwise, but only by running into a breakpoint.

When setting a breakpoint in a *parallel region*, turn on **All** in the **Ambiguous Line** dialog.

7 Programming tools

This chapter describes tools that are available to help you assess the performance of your code, identify potential performance problems, and locate the part of the code where most of the execution time is spent.

7.1 Sampling Collector and Performance Analyzer

The Sampling Collector and the Performance Analyzer are a pair of tools that you use to collect and analyze performance data for your application.

The Collector gathers performance data by sampling at regular time intervals and by tracing function calls.

The performance information is gathered in so called experiment files, which can then be displayed with the **analyzer** GUI or the **er_print** command after the program has finished.

7.1.1 The Collector

At first you have to compile your program with the

-g

option. Link the program as usual and then start the executable under the control of the Sampling Collector

collect collect_options a.out

Every 10 milliseconds profile data will be gathered and written in the experiment file

test.1.er

The number will be automatically incremented on subsequent experiments. In fact the experiment file is an entire directory with a lot of information. One can manipulate these with the regular Unix commands, but it is recommended to use the

er_mv, er_rm, er_cp

utilities to move, remove or copy these directories. This ensures for example that time stamps are preserved

After

er_print test.1.er

you can generate a first ASCII report from the experiment with the command

functions

Further commands are explained after invoking **help** or through the man page of the **er_print** command.

By selecting the options of the collect command, many different kinds of performance data can be gathered:

-p on off hi lo	Clock profiling ('hi' needs to be supported on the system)
-H on off	Heap tracing

-m on off	MPI tracing
-h counter0,0,counter1,0	Hardware Counters
-j on off	Java profiling
-S on off seconds	Periodic sampling (default interval: 1 sec)
-o experimentfile	Output file
-d directory	Output directory
-g experimentgroup	Output file group
-L size	Output file size limit [MB]
-F on off	Follow descendant processes

Various hardware counters can be chosen for collecting. Typing the collect command without any parameters, will print out all the counters available for profiling. Some of the events can only be gathered in register 0 and some only in register 1. Favorite choices are given in the following table.

-h cycles,0,instr,0	Cycle count, instruction count The quotient is the CPI rate (clocks per instruction) The optimum would be 0.25. The Mhz rate of the CPU multiplied with the instruction count divided by the cycle count gives the MIPS rate.
-h fpadd,0,fpmul,0	Floating point additions and multiplications The sum divided by the runtime in s gives the Mflop/s rate
-h cycles,0,dtlbm,0	Cycle count, data translation look-aside buffer (DTLB) misses A high rate of DTLB misses indicates an unpleasant memory access pattern of the program. Large pages might help (Solaris 9)
-h cycles,0,ecstall,0	L2 cache stall cycles.
-h cycles,0,dcstall,0	L1 plus L2 cache stall cycles
-h ecref,0,ecm	L2 cache references and misses
-h dcr,0,dcrm,0	L1 cache read references and read misses
-h dcw,0,dcwm,0	L1 cache write references and write misses

7.1.2 The Performance Analyzer

For the standard case of just collecting clock profiling and printing out the most important information in text mode a simple shell script is available:

```
$ /usr/local_rwth/bin/sample a.out
$ more sample.out
```



A program call tree with performance information can be displayed with the locally developed utility

```
$ /usr/local_rwth/bin/er_view
```



The full result of the analysis can be displayed graphically afterwards with the Performance Analyzer GUI, which has been redesigned in the latest version.

```
$ analyzer experimentfile.er
```

7.1.3 The Performance Tools Collector Library API

Sometimes it is convenient to group performance data in self defined samples, and to collect performance data only of a specific part of the program.

For this purpose the `libcollector` library can easily be used.

In the following example Fortran program, performance data only of the subroutines `work1` and `work2` is collected:

```
program test_collector
  call collector_pause()
  call preproc
  call collector_resume()
  call collector_sample("start")
  call work1
  call collector_sample("work1")
  call work2
  call collector_sample("work2")
  call collector_terminate_expt()
  call postproc
end program test_collector
```

The `libfcollector` library (C: `libcollector`) has to be linked. And if this program is started by

```
collect -S off a.out
```

performance data is only collected between the `collector_resume` and the `collector_terminate_expt` calls. No periodic sampling is done, but single samples are recorded whenever `collector_sample` is called. (The label is not currently used). When the experiment file is evaluated, the filter mechanism can be used to restrict the displayed data to the interesting program parts.

See the `libcollector` manual page for further information.

7.2 Frequency analysis with `tcov`

For error detection and tuning it might be helpful to know, how often each statement is executed. For testing a program it is important that all program branches are passed (*test coverage*). For this purpose, the program must be compiled and linked with the option

```
-xprofile=tcov
```

In the following program execution the frequencies of all statements recorded. The values can be entered in modified program sources using the command

```
$ tcov -a -50 -x a.out.profile \  
    [-p srcdir objdir] source_files...
```

Statements which have never been executed are marked by “#”.

7.3 Run time analysis with gprof

With **gprof**, a run time profile on module level can be generated. The program must be translated and linked with the option **-pg** (Fortran) resp. **-xpg** (C). During the execution a file named **gmon.out** is generated, which can be analyzed by

```
$ gprof program
```

With **gprof** it is easy to find out the number of the calls of a program module, which is a useful information for inlining.

NOTE: **gprof** assumes that all calls of a module are equally expensive, which is not always true. We recommend to use the Callers-Callees info in the Performance Analyzer to gather this kind of information. It is much more reliable.

7.4 Run time analysis of MPI programs

7.4.1 Sampling Collector and Performance Analyzer

With MPI programs, the Sampling Collector (see chapter 6.1) collects run time information for each MPI task, which can also be displayed for each task separately:

```
$ mprun -np n collect a.out
```

With a new option of the Sampling Collector MPI events can be traced as well

```
$ mprun -np n \  
    collect -m on -g experiment_group.erg a.out
```

together with the ability to bundle experiment files written by all MPI processes to experiment groups and display them with the Analyzer

```
analyzer experiment_group
```

Running collect with a large number of MPI processes, the amount of experiment data might become overwhelming. Starting the MPI program with a little script will help:

```
mprun -np 4 run.ksh
```

with

```
#!/bin/ksh  
# filename: run.ksh  
if [[ $MP_RANK < 2 ]]  
then  
    collect -m on -g test.erg a.out  
else  
    a.out  
fi
```

Performance information will be collected only for the MPI processes with rank 0

and 1

7.4.2 Prism environment

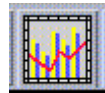
With the Prism debugger the communication structure of an MPI program can be traced.

```
$ prism -np n program
```

For Prism tracing of 32-bit programs the option **-32** must be added. Switch on the collection of run time information by clicking **Performance -> Collection** in the menu or by the input of **tnfcollection on** in the command line at the bottom. Start the program by clicking **run** or by input of the command **run**. An **xterm** window opens in which the program is started. When the program has completed, you can display the analysis with **Performance -> Display TNE Data** or by entering the command **tnfview**.

In the window that appears, you see the **Timeline** display, which shows the temporal behavior of the program.

After clicking the plot button



another window opens, in which you can produce and examine statistics for the individual calls of the MPI library. Select one of the displayed MPI calls and press the button **Create a dataset from this interval definition**. You have numerous options to vary the representation (don't forget to **refresh**).

7.4.3 Vampir and VampirTrace RWTH

Vampir is an MPI performance analysis tool sold by the company Pallas. After linking the **VampirTrace** library to the MPI program, a trace file is written during the program execution, which then can be displayed with the **Vampir** graphical user interface..

NOTE: The trace file produced at the end of a Fortran program run is always called **TRACE.bvp**.

Example in C:

```
$ . vampir.init
$ mpcc -o a.out ... *.c \
    -L/usr/local_rwth/lib -lVT -lmpi -lnsl
$ mprun -np 4 a.out
$ vampir a.out.bvp
```

Example in Fortran:

```
$ . vampir.init
$ mpf90 -o a.out ... *.f90 -R/usr/local_rwth/lib \
    -L/usr/local_rwth/lib -lVT -lmpi -lnsl
$ mprun -np 4 a.out
$ vampir TRACE.bvp
```

7.4.4 Jumpshot and the MPE Library

The Multi-Processing Environment (MPE) attempts to provide programmers with a complete suite of performance analysis tools for their MPI programs based on post processing approach. These tools include a set of profiling libraries, a set of utility programs, and a set of graphical visualization tools.

The most useful and widely used profiling libraries in MPE are the logging libraries. Various logfile formats are supported, the most powerful one is SLOG. As the default format is the CLOG, the programmer must set an environment variable to overwrite the default format:

```
$ export MPE_LOG_FORMAT=SLOG
```

After linking the libraries liblmpe.a (MPE logging interface) and libmpe.a (MPE graphics, logging, and other extensions) and, in the case of a Fortran program, the additional wrapper library libmpe_f2cmpl.a, the (binary) logfiles will be generated during runtime. Visualize these logfiles with the jumpshot (version 3) utility.

```
$ mpcc -c foo.c
$ mpcc -o foo foo.o \
    -L/usr/local_rwth/lib -llmpe -lmpe -lmpi
$ export MPE_LOG_FORMAT=SLOG
$ mprun -np 4 foo
$ jumpshot foo.slog
```

Example in Fortran:

```
$ mpf90 -c foo.f90
$ mpf90 -o foo foo.o -L/usr/local_rwth/lib \
    -lmpe_f2cmpi -llmpe -lmpe -lmpi
$ export MPE_LOG_FORMAT=SLOG
$ mprun -np 4 foo
$ jumpshot Unknown.slog
```

NOTE: The trace file produced at the end of a Fortran program run is always called **Unknown.bvp**.

8 Application software

8.1 Application software and program libraries RWTH

You will find the list of available application software and program libraries at <http://www.rz.rwth-aachen.de/sw/>

8.2 The Sun Performance Library

The Sun Performance Library is a part of the Sun One Studio Compiler Collection environment and contains highly optimized and parallelized versions of the well known standard libraries LAPACK version 3.0, BLAS, FFTPACK version 4 and VFFTPACK Version 2.1 from the field of linear algebra, Fast Fourier transforms and solution of sparse linear systems of equations (Sparse Solver, SuperLU) (see <http://www.netlib.org>). Please link your program with the options:

<code>-xarch=vplus8b -xlic_lib=sunperf</code>	32 bit addressing
<code>-xarch=v9b -xlic_lib=sunperf</code>	64 bit addressing

The performance of programs using the BLAS1-library can be improved by the new Fortran compiler option

```
-xknown_lib=blas
```

The corresponding routines will be inlined if possible.

The latest Performance Library contains new parallelized sparse BLAS routines for matrix-matrix multiplication and a sparse triangular solver. Linpack routines are no longer provided, it is strongly recommended to use the corresponding LAPACK routines.

Many of the contained routines have been parallelized using the shared memory programming model. Compare the execution times! Example:

```
$ f90 -dalign -mt -xlic_lib=sunperf ...  
$ ptime a.out  
$ ( export OMP_NUM_THREADS=4; ptime a.out )
```

The number of Threads used by the parallel Performance Library can be determined by the following call:

```
call USE_THREADS(n)
```

8.3 The Sun S3L library

The S3L-Library offers to MPI programs access to distributed arrays similar to the array descriptors, as they are used in the public domain packages ScaLAPACK and PETSc. The S3L-Library offers many functions from the fields linear algebra, Fourier transforms, etc. and further auxiliary functions (toolkit). Numerous kernel routines correspond to the ScaLAPACK interfaces.

The Toolkit functions are useful for working with parallel arrays and processor grids, as well as for parallel input or output. S3L arrays can be transformed into ScaLAPACK descriptors.

8.4 Nag Numerical Libraries [RWTH](#)

The Nag Numerical Libraries provide a broad range of reliable and robust numerical and statistical routines in areas such as optimization, PDEs, ODEs, FFTs, correlation and regression, and multivariate methods, to name but a few.

They are available in three flavours:

- 1) The serial NAG Mark 19 FORTRAN-Library (32 bit addressing mode)

```
f90 -xarch=v8plusb -dalign ... \  
-L/usr/local_rwth/lib -lnag19 \  
-xlic_lib=sunperf -lF77
```

- 2) The shared memory version, which includes 231 routines that benefit from shared memory parallelization (32- and 64-bit addressing modes) and has the identical programming interface as the serial version

```
f90 -dalign -xarch=v8plusb ... \  
-L/usr/local_rwth/lib -lnagsmp32 \  
-xlic_lib=sunperf -lF77  
  
f90 -dalign ... -xarch=v9b \  
-L/usr/local_rwth/lib -lnagsmp64 \  
-xlic_lib=sunperf -lF77
```

- 3) and the NAG Parallel Library Release 3.0, which contains 183 routines that have been specifically developed for use on distributed memory systems (32 bit addressing mode) using the MPI library.

```
mpf90 -dalign -xarch=v8plusb ... \  
-L/usr/local_rwth/lib -lnagmpi -ls31
```

Use the compiler version 6.2 to compile and link programs using the NAG libraries.

9 Further information

9.1 Sun products

9.1.1 on Sun's web site

Sun Online Dokumentation [Documentation] (<http://docs.sun.com>)

Sun Compiler Collection Portal (<http://forte.sun.com>)

Forte Developer 7 Collection

(<http://docs.sun.com/db?p=prod/fortedev7>)

Sun HPC ClusterTools (<http://docs.sun.com/software/hpc/index.html>)

9.1.2 on the RWTH Support Server

Forte Developer Programming Environment

(<http://support.rz.rwth-aachen.de/Manuals/SUN/WS6U2/html/home.html>)

Prism MPI Programming Environment

(<http://support.rz.rwth-aachen.de/Manuals/SUN/prism/html/home.html>)

Sun MPI Programming and Reference Guide (pdf)

Sun HPC ClusterTools Performance Guide (pdf)

Sun HPC ClusterTools User's Guide (pdf)

Prism Reference Manual (pdf)

Prism User's Guide (pdf)

Sun S3L Programming and Reference Guide (pdf)

9.1.3 on local file systems

Sun Online-Dokumentation zur Forte Developer Entwicklungsumgebung
(<file:///opt/SUNWspro/docs/index.html>)

Prism Entwicklungsumgebung

(<file:///opt/SUNWhpc/doc/prism/html/home.html>)

9.2 Third party products

TotalView (<http://www.etnus.com>)

KAP Pro/Toolset

(http://support.rz.rwth-aachen.de/Manuals/KAI/KAP_Pro_Reference.pdf,
<http://www.kai.com/parallel/kapro/index.html>)

Assure

(http://support.rz.rwth-aachen.de/Manuals/KAI/AssureT_Reference.pdf,
<http://www.kai.com/>)

Vampir and VampirTrace

(<http://support.rz.rwth-aachen.de/Manuals/Vampir/Vampir-userguide.pdf>,
<http://support.rz.rwth-aachen.de/Manuals/Vampir/VT20-userguide.pdf>,
<http://www.pallas.com>)

KCC

(http://support.rz.rwth-aachen.de/Manuals/KAI/KCC_docs/index.html,

http://www.kai.com/C_plus_plus/index.html)

Foresys (<http://www.simulog.fr>)

9.3 Public domain software

mpich – Eine portierbare Implementierung von MPI (<http://www-unix.mcs.anl.gov/mpi/mpich>)

PCL Performance Counter Library
(<http://www.fz-juelich.de/zam/PCL>)

9.4 Problems and inquiries

Helpdesk of the computer center (web interface)

10 Miscellaneous

10.1 Other Useful commands

<code>/opt/SUNWspro/bin/dmake</code>	Parallel make (compare gmake)
<code>/usr/bin/csplit</code>	Splits C programs
<code>/opt/SUNWspro/bin/fsplit</code>	Splits Fortran programs
<code>/usr/ccs/bin/nm</code>	Prints the name list of object programs
<code>/usr/bin/ldd</code>	Prints the dynamic dependencies of executable programs
<code>/opt/SUNWspro/bin/lint</code>	More accurate syntax examination of C programs
<code>/opt/SUNWspro/bin/cflow</code>	Prints the call hierarchy of a C program
<code>/opt/SUNWspro/bin/cxref</code>	Cross reference list of a C program
<code>/opt/SUNWspro/bin/ctrace</code>	Tracing of a C program
<code>/opt/SUNWspro/bin/dumpstabs</code>	Analysis of an object program
<code>/usr/bin/showrev</code>	Prints the software status of the machine
<code>/usr/bin/ptime</code> <code>/usr/bin/pstack</code> <code>/usr/bin/ptree</code> <code>/usr/bin/pmap</code>	Analysis of the /proc directory
<code>/usr/sbin/sysdef</code>	system parameters
<code>/usr/sbin/prtconf</code>	system configuration
<code>/usr/platform/SUNW,Sun-Fire/sbin/prtdiag</code>	diagnostic messages
<code>/usr/sbin/psrinfo</code>	processor information
<code>/usr/bin/pkginfo</code>	installed software packages
<code>/opt/SUNWspro/bin/fpversion</code>	processor information
<code>/usr/dt/bin/sdtprocess</code>	process list (compare top)
<code>/usr/bin/sar</code>	system activity report
<code>/usr/bin/truss</code>	log system calls
<code>/usr/bin/sotruss</code>	log of shared library calls
<code>ld.so.1</code>	Run time linker for dynamic objects
<code>/usr/bin/vmstat</code>	status of the virtual memory organization
<code>/usr/bin/iostat</code>	I/O statistics
<code>/usr/bin/busstat</code>	system bus performance counters
<code>gettimeofday</code> <code>#include <sys/time.h></code>	Portable real time counter