NEC SX-8 Cluster Documentation

Holger Berger

January 18, 2008

2

Contents

Chapter 1

Introduction

After HLRS offered access to SX-6 nodes for a year, the SX-8 system is now in operation. SX-8 and SX-6 are sharing a lot of characteristics, so SX-6 users should feel comfortable with SX-8 from the beginning. SX-8 offers with 72 nodes with 8 fast vector CPUs a so far unknown level of performance and throughput to HLRS users.

1.1   Hardware and Architecture

The system consists of two scalar frontend system, and 72 SX-8 nodes. The scalar frontend systems is a 32CPU Itanium2 system called asama, with a very large memory of 240 GB and a scalar frontend systems called a1 with 32CPU Itanium2 and 512GB of memory. Each CPU has a large 6MB L3 cache.

The frontend systems and the backend systems share fast filesystems. A total of two 9TB filesystems is used for user homes. One 80TB filesystem contain workspace which can be used during runtime of jobs. The workspace filesystem can sustain about 10GB/s in aggregate, for a single node about 600MB/s are possible.

The backend systems are 72 SX-8 vector nodes, each having 8 CPUs of 16 Gflops peak (2Ghz). Each node has 128 GB, about 124 GB are usable for applications.

The vector systems have a fast interconnect called IXS, which is a central crossbar switch. Each node can send and receive with 16 GB/s in each direction. You can expect an MPI latency around 5 microseconds for small messages. Each SX-8 CPU can access the main memory with 64 GB/s.

Features:

- Cluster of 72 SX-8 nodes

- Frontend is two times 32way NEC TX-7

- Batchsystem: NQSII

- Operating System: TX7: SUSE SLES9, SX8: SUPER-UX 15.1

1.2   Access

All former SX-6 users are registered for SX-8 and new frontend TX-7 *a1.hww.de*. Password was copied from crossi.hww.de. The a1.hww.de is integrated in the HLRS LDAP concept, so if you change the password on any other LDAP machine, it is reflected on a1.hww.de as well.

The only way to access the SX-8 nodes is through the TX-7 frontend using ssh to `asama.hww.de` or `a1.hww.de` (ssh) Information on how to set up ssh can be found on our webserver at http://www.hlrs.de/hw-access/platforms/ssh.html.

1.3   Main differences for SX-6 users

- new home directory

- small changes to mandantory NQSII limits

1.4   Usage

Interactive usage is allowed only on two systems. Prefered usage is on a1, where one can cross-compile for SX-8 and submit and monitor batch jobs.

To test executables, one can login to `v00.hww.de` from a1 using *rsh*. Thre are user limits in place limiting memory and cpu usage, and only 4 CPUs are available for interactive parallel tests.

Prefered usage of the system is through the batch system.

1.4.1   HOME directories

Users homedirectories and the global scratch directories are located on two fileservers in a fail-over configuration. Please note that the .profile in your home is for ksh as it is used on SX, and setup for VI editing mode. You might want to create a *.bash_profile* containt *set -o emacs* in your home to get normal bash command line editing on asama.

Default startup files (.profile,...) for your environmental settings can be found in:

`/usr/local/skel`

Only these startupfiles support the HWW cluster features like MPI, Compiler settings,...(see Program Development chap. **??** and Environment Settings chap. **??**).

Please also note that *tcsh* is not supported as a login shell.

1.4.2   SCRATCH directories

Different from SX-6 cluster, there is no default `SCRDIR` variable anymore.

You should still run your job in a working directory, but are now responsible to obtain it from the system.

Therefore, a mechanism called *workspaces* is implemented, which allows you to keep data outside your home not only during a run, but also after a run.

The idea is to allocate disk space for a number of days, and giving it a name, which allows you identify a workspace, and to distinguish several workspaces.

To get a new workspace, use the following line in your batch script:

`SCR='ws_allocate SimulatesomeThing 10`

`SCR` will contain the name of a directory which exists for 10 days, is on one of the temporary (faster) filesystems, and is owned by the caller.

The directory is not deleted after the job, but after 10 days of realworld time. In a second job, you can just use the same line to get the same directory. Please not that the directory of the example will be deleted 10 days after first usage, no matter how often it is used and what duration was specified in the subsequent calls.

If you want to monitor the progress of a batch job, you can find out the name of the workspace on a1.hww.de with the command `ws_list`, which lists all workspaces, their names and locations as well as remaining live time.

1.5 Filesystem Policy

IMPORTANT! NO BACKUP!! There is NO backup done of any user data located on HWW systems. The only protection of your data is the redundant disk subsystem. This RAID system (Raid3) is able to detect a failure of one component (e.g. a single disk or a controller). There is NO way to recover inadvertently removed data. Users have to backup critical data on their local site!

The homedirectory of each user is available on all nodes. A workspace is also available on all nodes and the frontend. To make most effective use of your workspace, try to do large, non-formatted I/O. Prefered blocksizes are larger than 4 MB per I/O request. (background: the filesystems are striped over 8 raid units each. To make best usage of the disk arrays, I/O has to be large)

1.6 Support/Feedback

Please report all problems to:

- System Administrators:

    - Bernd Krischok
      Phone: +49-711-685-87221
      e-Mail: krischok@hlrs.de

    - Thomas Beisel
      Phone: +49-711-685-87220
      e-Mail: beisel@hlrs.de

- Onsite NEC Support:

    - Holger Berger
      Phone: +49-711-685-87257
      e-Mail: hberger@hpce.nec.com

    - Danny Sternkopf
      Phone: +49-711-685-87256
      e-Mail: dsternkopf@hpce.nec.com

Chapter 2

Running Application

This chapter describes how to compile an application for SX-8 and the frontend system asama, a1, how to run an application, and how to create and monitor a batch job running on the vector system.

2.1   Compiling for SX

Remark for SX-6 users: SX-6 binaries will work on SX-8, but recompilation is recommanded anyhow to get best performance. SX-8 binaries will not work on SX-6.

Prefered way to compile for SX is usage of the crosscompilers on a1. The compilers running on a1 are a lot faster, and do not vaste expensive vector CPU resources for a non-vectorizable task.

For Fortran77 and Fortran90 codes, use `sxf90` command, for C and C++ codes, use `sxcc` and `sxc++` commands. If you create libraries, use the `sxmake` and `sxar` commands.

For MPI applications, use `sxmpif90`, `sxmpicc` and `sxmpic++`. Those commands alreday link the right MPI library, no further path or library specification is necessary.

For detailed information about compilers, refer to the online manuals http://www.hlrs.de/hw-access/platforms/sx8/doku/

Important note: The executables created by those compilers can not be executed on a1. Please login to v00 to test those executables, or submit a batch job.

To executa an MPI executable on SX, please use the `mpirun` command on the SX v00. Please see section about batch processing how to use it in batch scripts.

2.2   Compiling for TX7(Asama,a1)

Note: The TX7 are intented for usage like compilation for SX-8 (see previous section) and pre- and postprocessing of data for SX nodes. It is not foremost intended as a computation plattform on its own.

To create executables for TX7 Itanium CPU, use for Fortran77 and Fortran90 the `efc` command, for C++ and C the `ecc` command.

For MPI, use `mpif90` and `mpicc` commands. To run a MPI programm, use `mpirun` command on asama.

For detailed documentation about compilers, see on asama `/opt/NECcomp/compiler90/docs/c_ug_lnx.pdf` for C users guide and `/opt/NECcomp/compiler90/docs/for_ug_lnx.pdf` for a Fortran users guide. Please note that nice ftrace feature of SX is available on TX as well in slightly different form.

BLAS and Lapack can be found in `/opt/MathKeisan/lib`.

2.3   Creating Batch jobs

Asama, A1 and SX run one instance of the batch system NQSII, which is the follow on to the NQS which was used on SX-5.

Important note for SX-6 users: some new mandantory limits where introduced to garantuee smooth operations for all users. Please check your old batch jobs!

The basic policy is: 68 nodes are used for MPI multinode jobs, using more than 8 CPUs. Prefered are multiples of 8, like 16 or 32. Four nodes are used for other jobs. One node (v00) is used for interactive testing plus batch jobs. The four nodes run small test jobs, serial jobs and the smaller parallel MPI jobs and multithreaded jobs.

The batch system has two main entry points which do further sorting of the jobs. Use only those two entry points.

For all non-multi-node jobs, use the queue `dq`, for multinode jobs use `multi`. Queue can be specified on `qsub` command line using `qsub -q dq` or in the batch script. Please treat 8 CPU jobs as multinode jobs, throughput and wallclock times will be much better when running on a dedicated node.

2.3.1   Single-node jobs

Here is an example batch script for a normal single node job:

```
#!/usr/bin/ksh
#PBS -q dq                      # queue: dq  for <8 CPUs
#PBS -l cpunum_job=4        # cpus per Node              <!!!!!!!! NEW
#PBS -l cpunum_prc=4        # cpus per Node              <!!!!!!!! NEW
#PBS -b 1                       # number of nodes
#PBS -l elapstim_req=02:00:00 # max wallclock time     <!!!!!!!! NEW
#PBS -l cputim_job=02:00:00 # max accumulated cputime per job
#PBS -l cputim_prc=01:55:00 # max accumulated cputime per process
#PBS -l memsz_job=10gb      # max accumulated memory
#PBS -A <account code>       # Your Account code, see login message (without <>!)
#PBS -j o                       # join stdout/stderr
#PBS -N MyJob               # job name
#PBS -M jo@user                 # you should always specify your email address

SCR=`ws_allocate Run1 2`    # workspace for 2 days
cd $SCR
cp $HOME/code/mycode .
export OMP_NUM_THREADS=2     # use 2 OpenMP threads per MPI process, only for mult
export MPIPROGINF=YES     # give some performance information
export MPIMULTITASKMIX=YES  # we are hybrid openmp + mpi
export MPISUSPEND=ON      # be nice to others in shared mode, don't busy wait
mpirun -np 2 ./mycode      # start the code with 2 mpi processes
cp outfile \$HOME/code
# please be so kind to clean your workspace if files are not needed any more!
```

```
rm *.data                        # ... or whatever your files are called
```

This example shows a 4 CPU jobs, where two MPI processes spawn two threads each using OpenMP. The total CPU time is below 2 hours, and the jobs does not use more than 10 GB of memory. It is running on one node. The Jobname is for your convenience only, the account code should always be specified (otherwise, your jobs can not start!). After 2 hours, the job will be stopped due to elapsed time limit.

If you do not specify a CPU number, 1 CPU is assumed. Please note: The number of CPUs specified here has a meaning. It is used to find free resources by the batch system, and it is used to assign CPUs to your processes during runtime. But you still have to specify F_RSVTASK, OMP_NUM_THREAD or mpirun -np to really get this number of CPUs in the mix of processes and threads you want them to have.

A serial job can use up to 64 GB, and parallel job up to 124 GB.

2.3.2 Multi-node jobs

```
#!/usr/bin/ksh
#PBS -q multi                # queue: dq  for <=8 CPUs
#PBS -T mpisx                # Job type: mpisx for MPI
#PBS -l cpunum_job=8         # cpus per Node            !!!!!!!!!!!!!!!
#PBS -b 4                    # number of nodes
#PBS -l elapstim_req=02:00:00 # max wallclock time      <!!!!!!!! NEW
#PBS -l cputim_job=08:00:00 # max accumulated cputime per NODE
#PBS -l cputim_prc=07:55:00 # max accumulated cputime per process
#PBS -l memsz_job=10gb       # memory per node
#PBS -A <account code>       # Your Account code, see login message (without <>!)
#PBS -j o                    # join stdout/stderr
#PBS -N MyJob                # job name
#PBS -M jo@user              # you should always specify your email


SCR=`ws_allocate Run1 2`    # workspace for 2 days
cd $SCR
export OMP_NUM_THREADS=8
export MPIPROGINF=YES
export F_FILEINF=YES
export MPIMULTITASKMIX=YES
MPIEXPORT="OMP_NUM_THREADS F_FILEINF"
export MPIEXPORT
mpirun -nn 4 -nnp 1 ./mycode
cp outfile $HOME/code


# please be so kind and clean up your workspace from files no longer needed
```

This is a multinode example. Please note the directive `-T mpisx`, which is mandantory for multinode jobs!

This job used 4 nodes. It uses the first 4 nodes which are assigned by the batch system (-nn 4), one mpi process per node (-nnp 1). Each of them creates 8 OpenMP threads. Please note the usage of

MPIEXPORT to make sure that processes created on other nodes get the environment variables set by mpirun (it does NOT export all variables by default).

2.3.3   Batch Jobs on Scalar Frontends

You can also execute batch-jobs on the scalar frontends asama and a1. Those systems are scheduled by basic scheduler (see next chapter) based upon memory usage only.

```
#!/usr/bin/ksh
#PBS -q tx7                    # queue: tx7 for frontends
#PBS -l memsz_job=10gb         # memory usage
#PBS -j o                      # join stdout/stderr
#PBS -N MyJob                  # job name
#PBS -M jo@user                # you should always specify your email
```

2.4   Scheduling

There are three schedulers running, a simple batch scheduler, the so called extended resource scheduler ERS and the JobManipulator JMD.

The standard scheduler takes very small jobs (less than 10 GB, less than 600 seconds CPU time, less or equal 4 CPUs) and starts them in a queue called *test*. This queue can execute on the first four nodes, one job per node at a time. The jobs are served in the order they are submitted.

Jobs with less than 20 minutes walltime and small number of nodes (1 or 2) have good chances to get a quick start at nodes v04 and v05.

Medium sized jobs in the seriall and small-parallel queues (less than 8 CPUs) are scheduled by the ERS on the first 4 nodes, all other jobs (one node and more) are scheduled by JMD on nodes 6 to 71. ERS and JMD are a fair-share scheduler. Each user of the system has the right for the same share of the system. A user which is making heavy use of the system will get lower priority, to make sure other users can use as many resources, if they want.

Jobs are judged according to past usage of the system of the owner and resource request of the request. A smaller request is prefered, so it makes sense to make realistic resource allocations.

JMD is a backfilling scheduler, it tries to fill gaps created by larger jobs with smaller (shorter) jobs. Jobs with 16 or more nodes get special treatment, they can not be overtaken by other jobs of normal priority.

2.5   Monitoring the System

To monitor system usage, you can use qstat command of NQSII to see you requests, or you can use qs script on a1 to see all running and pending requests.

qstat output looks like:

```
RequestID        ReqName  UserName Queue      Pri STT S   Memory      CPU    Elapse
---------------- -------- -------- -------- ---- --- - -------- -------- --------
```

For detailed description of all fields, please see the qstat manpage on a1. *Jobs* shows the number of nodes a job requested. Please note that *CPU* time is cpu time of current running process within the job. If you want to see accumulated time of the whole request, use qstat -c 1.

To learn on which nodes your job is running, use `qstat -J`.

To see all jobs in the system, please use `qs` on a1 (not available on SX):

```
STAT REQ-ID  OWNER     NAME     QUEUE   NODES   TIME         TIME ESTIMATIONS
---- ------  --------  --------  --------  ---  ------------  ------------------------
```

qs shows the requests on the order as they will be started by ERS or JMD. For privacy reasons, you can not see all details of other users requests, but you can see all requests in the system, waiting or running.

The times and memory numbers show is current consumption and requested limit. The ESTIMATIONS colum gives the estimated time when the job will start, this estimations is based on a 72 hours prediction.

To delete a job, use the `qdel` command. Please note: qdel of NQSII does send a SIGTERM first, followed by a SIGKILL after 5 seconds. You can change the number of seconds using `-g` option. By using `-g -1`, SIGKILL is send immediatly.

Please avoid to write large stdout, please redirect stdout and stderr of you application into a file in your jobs directory. Writing large stdout requires spool space of unpredictable size, and always causes problems when trying to store back those files into users home directories.

Tip: If you want to make sure a batch request is able to clean up if it hits a time limit, specify a second limit. In addition to `cputim_job` you can specify a `cputim_prc`. Specify that limit a few minutes shorter, and the process hiting the limit (probably your simulation) will be killed first, and your batch job has some time to cleanup. Same applies to elapsed time limit.

Chapter 3

Program Development

this chapter is work in progress.

3.1 Libraries

3.1.1 MathKeisan (BLAS/LAPACK)

Blas and Lapack are in the standard library search path. Just use `-lblas` and `-llapack` to use it. See online manuals for other available tools in MathKeisan package.

Chapter 4

porting hints

4.1 porting to SX

This is a collection of pitfalls people use to get trapped when porting to SX.

- Code dumps core after C `malloc()` is used. Make sure you include `stdlib.h` when using `malloc()`, otherwise, return value is assumed to be *int* (C standard), and the SX calling conventions strips significant bits from the address. Make sure outcome of `malloc()` is never assigned to `int`, SX is LP64, not ILP64, a pointer does not fit into an int.

- Only 2GB of memory can be allocated with `malloc()`. For historical reasons, `size_t` is 32bit, as well as `sizeof(*void)` is 64bit. Use `-size_t64` for C or Fortran compilers to get rid of that restriction.

- My C code seems to have a problem with integer divisions. SX uses 56bit precision division for 64bit integer types per default. Use -xint switch to get full 64bit division (and loose some performance).

- Code stops with *Loop count is greater than that assumed by the compiler: loop-count=n* Compiler has sometimes to make a guess about loop length, to be able to allocate some work vectors (only for partially vectorized loops). This educated guess might be wrong. Help the compiler by giving *-W,-pvctl,noassume,loopcnt=n* where n is the maximum loop count, or a larger value.

- How are Fortran function names in the calling convention? Simply write them lowercase and append an underscore. Example:

```
PROGRAM TEST
CALL CFUNC(5)
END

void cfunc_(int *a) {
}
```

- What about parameters? Fortran is passing by reference, so use pointers in C for scalars. Character strings lead to an additional parameter of type long at the end of the parameter list, containing the length of the string. See C compilers manual chapter 4 for details.

- How to link when C and Fortran are mixed? Link with f90. He makes it right. When using C++ or using C++/SX as C compiler, use C++/SX as linker, using the option -f90lib.

4.2   Porting to IA64

- where are the fortran functions `etime, system, getarg` on Asama? Use `-Vaxlib` linker switch to link a compatibility library including those non-standard functions.

- is there a BLAS/LAPACK library on asama? Use `-L/opt/MathKeisan/lib/ -llapack -lblas` to use optimized versions of the libraries.

- where is documentation of the compilers? Search in `/opt/NECcomp/compiler70/docs` for `for_ug_lnx.pdf` and `c_ug_lnx.pdf`.

- How can I speed up fortran I/O? Set the `F_SETBUF=4096` to make the fortran library using large buffer size.

- why can I not read binary files written on SX on asama? SX is big-endian byte order (like SUN, IBM, SGI), Asama is little endian (like Linux x86 PCs). Fortran users can try to use endian conversion of the runtime library, search for `F_UFMTENDIAN` in either SX or Asama compiler manuals.

- How can I measure performance? You can use unix profiling by linking with `-p` and using gprof -b –no-graph to format the `gmon.out` file, or you can use `-ftrace` compiler switch, which will lead to some output after the program run. To redirect the output into a file, use `FTRACE_OUTPUT=filename`. Read the chapter about *simple performace analysis* of the compiler manuals for more information.

Chapter 5

Performance Tuning

5.1 Usage of hardware information

To get simple information about your application, please set F_PROGINF for Fortran or C_PROGINF for C/C++ programms. Possible values are YES and DETAIL. If this environment variable is set during runtime, an application prints out some timing information at programm end. Example F_PROGINF=YES:

```
    ******  Program Information  ******
 Real Time (sec)          :        26.804331
 User Time (sec)          :        26.275639
 Sys  Time (sec)          :         0.308973
 Vector Time (sec)        :        24.559803
 Inst. Count              :      2334758597.
 V. Inst. Count           :      1305590123.
 V. Element Count         :    271274447129.
 FLOP Count               :    129865319762.
 MOPS                     :     10363.348892
 MFLOPS                   :      4942.422871
 VLEN                     :       207.779181
 V. Op. Ratio (%)         :        99.622051
 Memory Size (MB)         :        48.031250

 Start Time (date)  :  2004/04/13 14:54:42
 End   Time (date)  :  2004/04/13 14:55:09
```

 Example F_PROGINF=DETAIL:

```
    ******  Program Information  ******
 Real Time (sec)          :        26.768830
 User Time (sec)          :        26.289378
 Sys  Time (sec)          :         0.309090
 Vector Time (sec)        :        24.571109
 Inst. Count              :      2334758675.
 V. Inst. Count           :      1305590123.
 V. Element Count         :    271274447129.
 FLOP Count               :    129865319762.
```

```
MOPS                          :            10357.932794
MFLOPS                        :             4939.839858
VLEN                          :              207.779181
V. Op. Ratio (%)              :               99.622051
Memory Size (MB)              :               48.031250
MIPS                          :               88.809961
I-Cache (sec)                 :                0.041723
O-Cache (sec)                 :                0.059745
Bank (sec)                    :                0.000690

Start Time (date)  :  2004/04/13 14:56:24
End   Time (date)  :  2004/04/13 14:56:51
```

Using this variable does only cause constant overhead (reading counters at the beginning, reading at end and computing and priunting of values).

General strategy for tuning is:

vector time should be as close at possible to *user time*. This means, *V. Op. Ratio* will be close to 100. *MFLOPS* should be as high as possible (as long as the application is doing floating point operations). A value between 2000 and 4500 is respectable. If it exceeds 9000, celebrate a miracle.

To achieve good performance, *O-Cache* (operand cache misses in seconds) should be close to 0. Vectorized code can not cause o-cache misses! If *V. Op. Ratio* is 99, but performance in MFLOPS is still bad, there are several possibilities:

- no floating point operations in the code

- short vector length *VLEN*

- high *bank* times

VLEM is the average vector length which is processed by vector pipes. So this is average of $(looplength) modulo 256$, and can therefor not exceed 256, no matter how long loops are. It should be close to 256. The longer the loops are, the more efficient the CPU can work. Try to achieve loop length in the order of thousands.

High *bank* times show a high number of bank conflicts. A bank conflict is caused when a memory bank is accessed before the bank busy time from the last access is over. If this is high, search for power of two leading dimensions in the code. Distances between memory accesses in the form of a multiple of a large power of two should be avoided. Try to have odd or prime distances, best is stride 1 (in unit of words). When using lookup tables, high bank conflict times can arise as well if the lookup always hits the same value (what might be a consequence of cache optimizations). Try to make copies of the tables, and iterate over the tables.

5.2   Profiling on subroutine Basis

Simple Unix profiling by linking the code with `-p` option and calling `prof <executable name>` processing the generated `mon.out` file does not cause large overhead. This gives you simple information how much time was spent in which subroutine.

If yoy want to know the number of calls of the subroutine as well, the subroutines have to recompiled with `-p` as well. This causes more overhead.

5.3 Hardware information on subroutine Basis

To get more detailed information about subroutines, use `ftrace` feature of the compilers.

Compile all subroutines you want to examine, but at least the entry and the exit of your application with `-ftrace` (Fortran and C/C++).

Run the application, and keep the generated *ftrace.out* files. Use the `sxftrace/ftrace` tool to get the actual information out of the binary file.

Example output:

```
*-------------------------*
  FLOW TRACE ANALYSIS LIST
*-------------------------*


Execution : Tue Apr 13 15:33:11 2004
Total CPU : 0:00'29"189



PROG.UNIT   FREQUENCY  EXCLUSIVE        AVER.TIME   MOPS MFLOPS V.OP  AVER.    VECTO
                       TIME[sec]( % )    [msec]                  RATIO V.LEN    TIME

chempo           1000  13.452( 46.1)    13.452 8832.9 4631.5 99.54 217.1    11.18
kraft            1100  13.410( 45.9)    12.191 11377.9 5013.2 99.81 201.0    13.2
zufall        4324320   2.181(  7.5)     0.001  148.7    5.9  0.00   0.0     0.00
lj2                 1   0.059(  0.2)    59.033  328.0  173.3 85.06 237.9     0.00
korekt           1100   0.041(  0.1)     0.037 7792.8 3285.9 99.71 222.8     0.03
voraus           1100   0.035(  0.1)     0.032 8537.4 3976.9 99.76 242.1     0.03
transf           1100   0.005(  0.0)     0.005 11972.9 5976.8 99.84 216.0     0.0
skal             1100   0.004(  0.0)     0.004 3554.4 1170.5 98.09 240.0     0.00
geschw              1   0.002(  0.0)     1.726  161.4   30.3 12.72 230.0     0.00
gitter              1   0.000(  0.0)     0.071  333.2   43.3 21.77 235.6     0.00
init              100   0.000(  0.0)     0.000  249.3    0.1 43.79 235.6     0.00
psi22               4   0.000(  0.0)     0.004  110.8   15.1  0.00   0.0     0.00
phi22               4   0.000(  0.0)     0.002  112.9   15.3  0.00   0.0     0.00
cor22               2   0.000(  0.0)     0.004   86.7    8.6  0.00   0.0     0.00
cutoff              1   0.000(  0.0)     0.002   89.4    5.3  0.00   0.0     0.00
-------------------------------------------------------------------------------
total         4330934  29.189(100.0)     0.007 9333.6 4449.1 99.57 207.8    24.55
```

5.4 Hardware information on loop basis

If you need to have more detailed information of the contents of a subroutine, regions within subroutines can be defined to be used with ftrace.

Enclose the section to be examined by special ftrace function calls:

```
        CALL FTRACE_REGION_BEGIN("REGION_A")
        DO I=1,10000
```

```
          A(I)=I
       ENDDO
       CALL FTRACE_REGION_END("REGION_A")
```

   or in C/C++:

```
ftrace_region_begin("region_a");
/* region */
ftrace_region_end("region_a");
```

   The name is a name you can choose, it has to match in the begin and end call. This will be used
as the identifier in the ftrace printout.

5.5   I/O

Because underlaying GFS uses striping, large I/O should be done to make efficient use of the re-
sources.

   Try to make I/O in multiple of 4MB blocks, if this is possible. For fortran, setting the units I/O
buffer to 4 MB improves I/O speed a lot. Use `export F_SETBUF<unit>=4096` to set buffersize
for fortran unit `<unit>`.

   To assist in tuning of fortran I/O, it is possible to generate statitics of the I/O (only fortran). Set
`export F_FILEINF=YES` or `export F_FILEINF=DETAIL` to get information about I/O sizes,
achieved transfer bandwidth and settings of the fortran units.

   In C, use `setvbuf` to increase the buffer size of buffered I/O calls `fwrite` and `fread`. Call
`setvbuf` after `fopen` but before first actual I/O.

   In C++, use `pubsetbuf` like in this example:

```
#include <fstream>
using namespace std;

int main(int argc, char **argv)
{
        char buffer[4096*4096];
        fstream out_file("huhu",ios::app|ios::out);
        out_file.rdbuf()->pubsetbuf(buffer,4096*4096);
        out_file << "Hallo" << endl;
}
```

5.5.1   example timings of Fortran I/O

The following timings on Asama and SX show, that formatted I/O is very slow, and any kind of
unformatted I/O is faster. Example shows the effect of setting `F_SETBUF=4086` as well. Idea is to
write on a 2D array without the border. One can see that writing out the array with the border is the
fasted. Code is following little test:

```
program fio

include 'mpif.h'
```

```
real*8 :: feld(5000,5000)
real*8 :: buffer(4998,4998)
real stimes(2)
real etimes(2)
real d
real*8 time1,time2

call MPI_Init(ierr)

open(10,file='test.dat')
print *,'write(10,*) feld'
time1= MPI_wtime(ierr)
d= etime(stimes)
write(10,*) feld
time2= MPI_wtime(ierr)
d= etime(etimes)
print *,etimes(1)-stimes(1),etimes(2)-stimes(2),time2-time1
close(10)

open(10,file='test.dat',form='unformatted',status='replace')
print *,'write(10) feld'
time1= MPI_wtime(ierr)
d= etime(stimes)
write(10) feld
time2= MPI_wtime(ierr)
d= etime(etimes)
print *,etimes(1)-stimes(1),etimes(2)-stimes(2),time2-time1
close(10)

open(10,file='test.dat',form='unformatted',status='replace')
print *,'write(10) feld(2:4999,i)'
time1= MPI_wtime(ierr)
d= etime(stimes)
do i=2,4999
  write(10) feld(2:4999,i)
end do
time2= MPI_wtime(ierr)
d= etime(etimes)
print *,etimes(1)-stimes(1),etimes(2)-stimes(2),time2-time1
close(10)

open(10,file='test.dat',form='unformatted',status='replace')
print *,'write(10) buffer'
time1= MPI_wtime(ierr)
d= etime(stimes)
do i=2,4999
```

```
   buffer(:,i-1)=feld(2:4999,i)
end do
write(10) buffer
time2= MPI_wtime(ierr)
d= etime(etimes)
print *,etimes(1)-stimes(1),etimes(2)-stimes(2),time2-time1
close(10)

call MPI_Finalize(ierr)

end program fio
```

Printed timing is user time, system time and wallclock time.

```
asama, without F_SETBUF
 write(10,*) feld
   20.16895        2.594727        78.4661729335785
 write(10) feld
   3.906250E-02   0.8447266       15.8332600593567
 write(10) feld(2:4999,i)
   6.835937E-02   1.784180        36.0356490612030
 write(10) buffer
  0.2294922        0.9208984       17.2316899299622

asama, with F_SETBUF=4096
 write(10,*) feld
   20.55859        5.859375E-02   24.3654959201813
 write(10) feld
  0.2050781        1.171875E-02   1.03501415252686
 write(10) feld(2:4999,i)
  0.2031250        8.007813E-02   7.26763582229614
 write(10) buffer
  0.2695313        0.1474609       1.13990497589111
```

So by avoiding formatted I/O and setting the buffer size, it is possible to speed up I/O by a factor of 70 on Asama.

```
SX-6, without F_SETBUF
 write(10,*) feld
 83.49000  39.87500  168.8392629427835
 write(10) feld
 0.000000E+00  4.499816E-02  0.5913347122259438
 write(10) feld(2:4999,i)
 0.1600037  7.735001  21.64797224197537
 write(10) buffer
 1.499939E-02  4.000091E-02  0.6230526024010032

SX-6, with F_SETBUF=4096
```

```
write(10,*) feld
81.13000  3.790000  99.40329434094019
write(10) feld
0.000000E+00  3.999996E-02  0.5728084549773484
write(10) feld(2:4999,i)
6.000518E-02  0.1949999  1.330135225551203
write(10) buffer
9.994506E-03  4.500007E-02  0.5996705272700638
```

On SX-6, a speedup of 280 can be achieved.

Copying data in one buffer and writing that buffer with one I/O call is faster on both machine than using small I/O with array syntax.

5.6 Psuite

5.6.1 gui mode

Section to be written, please refer for the moment beeing to psuite manual of the online documetation.

5.6.2 command line interface

Here is a short starter guide how to use the command line interface of psuite.

- compile and link executable with `-pspa` option (not listed in compiler manuals), which will allow further measurements, but causes some overhead (not for production use!)

- start `pspfl <executable>`

- select where to measure, e.g. `(PSperf) select routine all`

- choose what to be measured, e.g. `(PSperf) collect cpu elapse instructions iteration memory`

- save the information into the executable: `(PSperf) save executable`

- run the executable on the SX

- get the .pdf file

- tell psuite the name of the pdf file: `(PSperf) set pdf <pdfname>`

- to get reports, do `(PSperf) analyze`

To select loops in a subroutine, give a `select loop in <sub>`. This selects all outermost loops in the subroutine.

If you want all reports to be sorted according to CPU time usage, do a `export PSPERF_USEPARC=YES` before starting `pspfl`.

To store a report into a file, just use `analyze ><filename>`.