CENTORI - Installation, Performance, and Enhancements

Georg Huhs

Barcelona Supercomputing Center

Technical Report TR/CASE-10-1

Dec 2010

CENTORI - Installation, Performance, and Enhancements

Georg Huhs[1]

[1] *Computer Application in Science & Engineering, Barcelona Supercomputing Center, Edifici Nexus, Campus Nord UPC, c/ Gran Capità, 2-4, 08034 Barcelona, Spain.*
E-mail: `georg.huhs@bsc.es`, *Web page:* `http://www.bsc.es`

Abstract. This report shows the proceedings for installing and running the code CENTORI at MareNostrum, includes some performance analysis and describes the implementation of parallel IO done at BSC.

Key words. CENTORI , installation guide, execution guide, parallel IO

Contents

1. Introduction

CENTORI is an electromagnetic turbulence simulation code for two-fluid tokamak plasma turbulence as would be found in MAST, JET, and ITER.

This report shows the work on CENTORI done at BSC. It consists of three parts: porting CENTORI to MareNostrum (sections 2 and 3), performance analysis (section 4), and introducing parallel IO using MPI-IO functionality (section 5).

2. Compilation

2.1. Prerequisites

CENTORI does not use external libraries and follows Fortran 95 standard. Thus the only prerequisite is a recent F95 compiler.

2.2. Compiling at Mare Nostrum

CENTORI is deployed as tar.gz file, which has to be extracted

```
tar -xzf filename.tar.gz
```

into the destination folder.

There is no config-script, all configuration is done directly in makefile. In order to compile CENTORI at Mare Nostrum, following lines have to be added to the makefile into the section titled "architecture specifics":

```
###### Mare Nostrum - Intel

F90_MN     = mpif90
FFLAGS_MN = -q32 -qsuffix=cpp=f90 -O3 -qtune=ppc970 -qarch=ppc970 \
            -qcache=auto -I/gpfs/apps/FFTW/3.2.1/32/include
LFLAGS_MN = -q32
ifeq (${FFTW},YES)
        LIBS_MN    = -L/gpfs/apps/FFTW/3.2.1/32/lib -lfftw3
endif
```

The recommended compiler option -qstrict causes problems during compilation and is not used for this reason.

Further the variable ARCH, which selects the "architecture" to use, needs to be set:

```
ARCH = MN
```

In order to use MPI2, environment variables need to be set correctly before compiling, for example by executing

```
export PATH=/gpfs/apps/MPICH2/mx/1.0.7..2/bin/:$PATH
export MP_IMPL=anl2
```

The compilation is started by calling

```
make
```

2.3. Generating documentation

Two types of documentation are provided:

- The LaTeX file centori.tex (together with some pictures) containing a description of the mathematical background.
- An "autodoc" mechanism, that extracts information from appropriate comments in the source code and creates a set of HTML files, one for each module and subroutine.
  It is a Fortran application (autodoc.f90) which reads the data it has to process from the standard input.

The makefile provides mechanisms for generating both by the three commands

`make latex` compiles the LaTeX file (only to dvi)

`make autodoc` compiles only autodoc.f90 and does not generate the documentation

`make doc` executes the above two commands and generates the documentation. It feeds autodoc with all source files. Since LaTeX is not installed on Mare Nostrum's login nodes, this operation fails when using one of them. One has either to use another computer or to alter the makefile to execute only autodoc.

3. Running CENTORI

3.1. Input files

CENTORI needs the following input files:

- centori.in
- grass.in
- datasets.dat

Concerning parallelisation the most interesting one is centori.in, since it contains variables specifying the partition of the domain and thus the number of processors needed. These variables are NX_SPROC, NY_SPROC, and NZ_SPROC. The resulting number of processors equals NX_SPROC ∗ NY_SPROC ∗ NZ_SPROC.

3.2. Executing CENTORI

The executable centori.exe does not need any additional command line arguments. Only the path to the MPI lib has to be set.

A valid execution script is:

```
#!/bin/bash
# @ job_name          = centori
# @ initialdir        = .
# @ output            = centori.out
# @ error             = centori.err
# @ mpi2              = 1
# @ total_tasks       = 32
# @ cpus_per_task     = 1
# @ wall_clock_limit  = 01:00:00

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/gpfs/apps/MPICH/mx/32/lib/
export OBJECT_MODE=32

date
time srun ./centori.exe > centori.out
date
```

When using MPI1, the variable mpi2 and the variable LD_LIBRARY_PATH have to be altered:

```
...
# @ mpi2              = 0
...
#export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/gpfs/apps/MPICH/mx/64/lib/
...
```

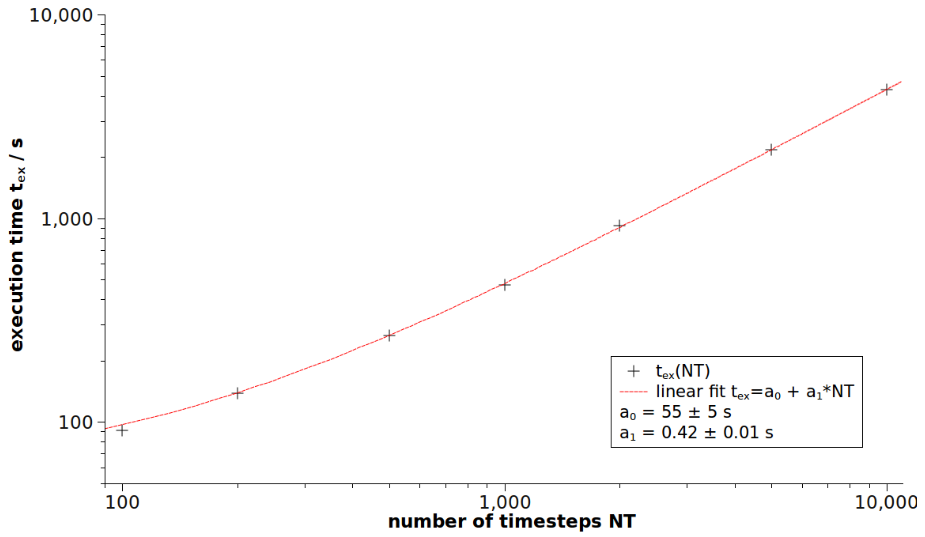

Figure 1. Simulation running time depending on the number of simulated timesteps

4. Performance analysis

The analysis shown here has been done with CENTORI version svn393 before any of the changes described in chapter 5. It is based on the configuration file given in listing 1 (appendix A.1). Relevant changes to this configuration are specified when necessary.

4.1. Computational effort depending on the number of timesteps

The computational effort scales, as one may expect, linearly, with an offset of 55 ± 5 seconds. For details see figure 1. These simulations used 32 processors.

4.2. Scalability

As figure 2 shows, CENTORI scales moderately up to 128 processors. Beyond that no more speedup is achieved. When running with higher NRGRID and NZGRID similar behaviour shows up.

To vary the number of processors used, the parameters NX_SPROC, NY_SPROC, and NZ_SPROC were set to the following values:

#processors	NX_SPROC	NY_SPROC	NZ_SPROC
1	1	1	1
16	4	2	2
32	4	4	2
64	4	4	4
128	8	4	4
256	8	8	4

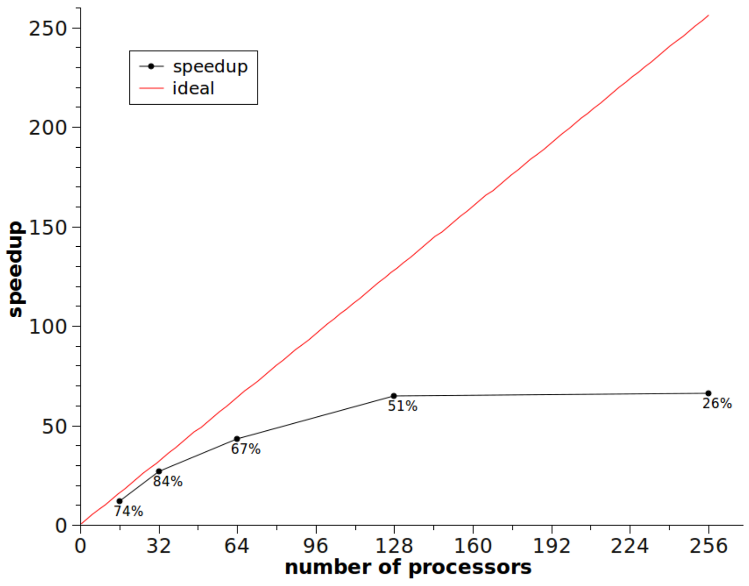

Figure 2. Speedup depending on the number processors

5. Implementation of parallel IO using MPI

5.1. Introduction

Before the introduction of parallel IO, similar output files were written by each process. The only exception were the files written by process 0, who contained some additional information.

Writing data of all processes into collective files would have, besides the obvious effect of reducing the number of files written and so enhancing clarity, several advantages, depending on the type of output file.

There are three kinds of output files:

save files contain data for a warm start. With a single file the simulation can be continued with any number of processes (suitable for the simulation). The collective save file introduced is named save_coll .

dump files contain, amongst others, data for visualisation. With a single file containing all data the whole domain can be visualised easily. There are two collective dump files generated: dump_coll and dump_coll_f which contain the same data, the first one in primitive binary form, the latter one in Fortran format.

out files are used for writing diagnostics into. Since it makes sense to keep this processor dependent and these files are not as important as the data files, they will not be parallelised.

Every NT timesteps the output is written by generating a new save file and appending the current time dependant data to the dump file.

Since save and dump files contain the same type of data, there is no difference between them concerning parallelisation. Thus from now on both will often be referred to as data files.

In principle the introduced collective data files should equal files created by running the non parallel routines with only one processor. In fact there are differences, because Fortran's build in write commands enclose each record with some markers, in contrast to the MPI write commands, which write only the provided data. This is no problem for the save-files, because reading them is programmed in a convenient way. But the dump file is read by external programs, so it has to feature exactly the same structure as the old one. This is achieved by generating a common dump file and converting it to the Fortran format afterwards.

5.2. Types of data

The data contained in the data files can be categorized by the number of dimensions it spans. Each category needs different treatment.

Common data has no spatial dependency, which means that all processes posses the same data. These are vectors containing the partitioning of the spatial axis (grid) and single values, e.g. plasma current, total energy, ...

Profiles depend only on one dimension, which is always the ψ-direction.

R,Z grid functions depend on R and Z. This data is not split amongst the processes.

Planes are two dimensional functions depending on ψ and θ

Fields are three dimensional functions (depending on ψ, θ, and ζ), which may be scalar or vectorial.

5.3. Structure of data

Spatial data is split amongst a 3D processor grid. The data a single process holds, called local data, consists of the core region, resulting from simply slicing the global data, and a surrounding halo, whose data is taken from spatially adjacent regions. The resulting duplication of data needs to be taken care of when reading and writing it.

The datastructures for handling this issue are set up in

```
subroutine initialise_parallel_io
```

The most important parameters are the local halos, read/write sizes, and read/write starts. The starts are zero based indices specifying a position inside the global data grid.
Further the filetypes needed by the fileviews for the MPI read and write commands are defined in this routine.

5.4. Array storage format

Arrays are stored in Fortran's column-first order.

5.5. General implementation notes

Data is read/written either by using collective routines and individual file pointers (for planes and fields) or non collective ones and explicit offsets (all other data). For calculating the correct offset, two variables are used:

`file_pos` which defines a the starting point in the file where the next read/write should start. Thus it has to be updated after each operation. If it is handed over to a subroutine, it is their responsibility to do the update.

`disp` which is the displacement of the process specific segment of the data inside the current block of data. This is needed only when using explicit offsets

5.6. Writing data

The basic writing routine for the save file is

```
subroutine write_parallel_save_file
```

It defines the order of the variables to write. It is also responsible for selecting the data to be written by the main process only and keeping track of file_pos .
Since the save file reflects only the data at a single point in time, there is a new save file generated at each call of write_parallel_save_file .

A very similar routine generates the common dump file:

```
subroutine write_parallel_dump_file(output_no)
  ! Arguments
  integer, intent(in) :: output_no
```

Its structure is similar to write_parallel_save_file with two main differences. Firstly it has to check which data should be written to the dump file (specified in the file datasets.dat). Secondly it also organises the handling of time dependant output. Time independant output is written in the first call of this routine, followed by the time dependant data, which is appended at each execution of write_parallel_dump_file .

5.6.1. Common data

Common data is written by the main process only. Single values are collected into a vector, which is written at once. Vectors are processed by a single write for each.

5.6.2. Profiles

Profiles are split amongst the processor grid x-slab. Thus it would be sufficient if only processors that lie on one specific slab write their portion of data, but the collective MPI-IO functions are used because they perform much better.

For writing profiles given in different formats an overloaded subroutine structure that writes a local profile's data into the file has been introduced:

```
interface write_profile_parallel
  module procedure write_profile_parallel__sca_prf
  module procedure write_profile_parallel__array
end interface

subroutine write_profile_parallel__sca_prf(file_handle, local_prf, file_pos)
  ! Arguments
  integer,                          intent(in)    :: file_handle
  type(sca_prf),                    intent(in)    :: local_prf
  integer(kind=MPI_OFFSET_KIND),    intent(inout) :: file_pos

subroutine write_profile_parallel__array(file_handle, local_prf, file_pos)
  ! Arguments
  integer,                                       intent(in) :: file_handle
  real(kind(1.0D0)), dimension(0:npsi_sub),      intent(in) :: local_prf
  integer(kind=MPI_OFFSET_KIND), intent(inout)              :: file_pos
```

5.6.3. R,Z grid functions

The main process writes these two dimensional fields by a single operation.

5.6.4. Planes (ψ, θ grid functions)

A plane is split amongst a processor-grid-plane. Thus only the processes of a single processor-grid-plane would need to perform the writing, but, as with profiles, a much better performance is achieved when all processors use a collective MPI-IO function. The routine write_plane_parallel does the writing. Its signature is:

```
subroutine write_plane_parallel(file_handle, local_pln, file_pos)
  ! Arguments
  integer,                                  intent(in)    :: file_handle
  real(kind(1.0D0)), dimension(0:npsi_sub, 0:ntheta_sub), &
                                            intent(in)    :: local_pln
  integer(kind=MPI_OFFSET_KIND),            intent(inout) :: file_pos
```

5.6.5. Fields

There are scalar and vectorial fields. A **vector field** is written by splitting it up in three scalar fields, corresponding to the components of each vector, and writing them in sequence. This is done by the routine

```
subroutine write_vector_field_parallel(file_handle, local_vec_fld, file_pos)
  ! Arguments
  integer,                           intent(in)    :: file_handle
  type (vec_fld),                    intent(in)    :: local_vec_fld
  integer(kind=MPI_OFFSET_KIND),     intent(inout) :: file_pos
```

A (global) **scalar field** is stored in a file as a contiguous block of data, resulting from reordering the 3D data in Fortran style. Each process writes its portion of data using a fileview, created with the help of MPI_TYPE_CREATE_SUBARRAY. In doing so, a process does not write all of its local data to avoid writing overlapping data twice. Responsible for handling a scalar field is the routine

```
subroutine write_field_parallel(file_handle, local_field, file_pos)
  ! Arguments
  integer,                           intent(in)    :: file_handle
  real(kind(1.0D0)), dimension(0:npsi_sub, 0:ntheta_sub, 0:nzeta_sub), &
                                     intent(in)    :: local_field
  integer(kind=MPI_OFFSET_KIND),     intent(inout) :: file_pos
```

A specialisation for writing to the dump file is

```
subroutine write_field_parallel_dump(file_handle, local_field, file_pos)
  ! Arguments
  integer,                           intent(in)    :: file_handle
  real(kind(1.0D0)), dimension(0:npsi_sub, 0:ntheta_sub, 0:nzeta_sub), &
                                     intent(in)    :: local_field
  integer(kind=MPI_OFFSET_KIND),     intent(inout) :: file_pos
```

It differs from write_field_parallel by the areas of the global data it processes, since for dump files no halo in the θ and ζ direction is written. Further the user can specify if all ζ planes shall be saved. If not, only the first non-halo plane is written to the dump file.

The collective MPI-IO routine performs much better than its non collective counterpart (order of magnitude 10 times faster).

5.7. Reading data

For reading from the file the same principles as for writing are used. There are two main differences:

- Data that is used by more than one processors may be broadcasted after being read by one processor.
- The sizes and start positions differ from those for writing. Each process reads all of its local data (including the halo) at once.

Reading concerns only the save file because the dump file is never read by Centori .

The main reading routine for the save file is

```
subroutine read_parallel_save_file
```

Of course its structure has to reflect the organisation of write_parallel_save_file .

5.7.1. Common data

Only process 0 reads the common data. It uses one read-command for each vector. The single values are read as vector in one step. Afterwards the vector's data is assigned to the appropriate variables. Distributing these data to the processes is done in the simulation-initialisation-routines, not in the data-reading-routine.

5.7.2. Profiles

Profiles are split amongst the processor grid x-slab, which offers two ways of reading them:

- Data is read only by processors that lie on one specific slab (selected by the lowest position in y and z direction of the processor grid) and broadcasted along the yz-plane afterwards.
- Every process reads its data directly from the file.

Both variants are implemented, broadcasting can be selected by *#define PROFILE_BCAST*.

For reading profiles a subroutine, that allocates a profile datastructure (**type**(sca_prf)) and reads its data from the file, has been introduced. Its signature is:

```
subroutine read_profile_parallel(file_handle, local_prf, file_pos)
  ! Arguments
  integer,                          intent(in)    :: file_handle
  type(sca_prf),                    intent(out)   :: local_prf
  integer(kind=MPI_OFFSET_KIND),    intent(inout) :: file_pos
```

5.7.3. R,Z grid functions

Similar to common data, only process 0 needs to read this data, which will be broadcasted during the simulation's initialisation.

5.7.4. Fields

When dealing with fields each processor reads all of its local data from the file. The structure for reading is very similar to writing.

Vector fields are read component-by-component into a temporary buffer, which is used for initialising a newly created vec_fld structure. This job is done by the routine

```
subroutine read_vector_field_parallel(file_handle, local_vec_fld, file_pos)
  ! Arguments
  integer,                          intent(in)    :: file_handle
  type (vec_fld),                   intent(out)   :: local_vec_fld
  integer(kind=MPI_OFFSET_KIND),    intent(inout) :: file_pos
```

Scalar fields are dealt with the following function, which reads all local data into a buffer.

```
subroutine read_field_parallel(file_handle, local_field, file_pos)
  ! Arguments
  integer,                          intent(in)    :: file_handle
  real(kind(1.0D0)), dimension(0:npsi_sub,0:ntheta_sub,0:nzeta_sub), &
                                    intent(out)   :: local_field
  integer(kind=MPI_OFFSET_KIND),    intent(inout) :: file_pos
```

5.8. Converting the dump file

One has to take onto account that the dump file is read by external programs, which expect it to be written in Fortran format, produced by using Fortran's build in write commands. They enclose each record with specific markers, in contrast to the MPI write commands, which write exactly the provided data - and nothing more.

The desired result is achieved by generating a collective dump file in a parallel way and converting it afterwards. The conversion is done by one processor, reading record by record from the existing dump file with MPI commands and writing the data to a new file using Fortran's write command.

This approach is not the most elegant one, but it is maintainabel because of the relative small additional effort. The usual test (see A.1) running on 32 processors and writing output only once took 658 seconds and produced a 6.5MB collective dump file. (The size of this file was maximised by writing all possible data to it.) Converting this dump file lasted about 0.3 seconds, which is only about 0.05% of the total running time.

This conversion is driven by the routine

```
subroutine convert_dump_file(num_outputs)
  ! Arguments
  integer, intent(in) :: num_outputs
```

Further there is a set of routines specialised on transferring several types of data:

```
subroutine transfer_int(from_file, to_file, file_pos)
  ! Arguments
  integer,                          intent(in)    :: from_file, to_file
  integer(kind=MPI_OFFSET_KIND),    intent(inout) :: file_pos


subroutine transfer_real(from_file, to_file, file_pos)
  ! Arguments
  integer,                          intent(in)    :: from_file, to_file
  integer(kind=MPI_OFFSET_KIND),    intent(inout) :: file_pos


subroutine transfer_string(from_file, to_file, num_characters, trim_string,
    file_pos)
  ! Arguments
  integer,                          intent(in)    :: from_file, to_file
  integer,                          intent(in)    :: num_characters
  logical,                          intent(in)    :: trim_string
  integer(kind=MPI_OFFSET_KIND),    intent(inout) :: file_pos


subroutine transfer_real_array(from_file, to_file, size, file_pos)
  ! Arguments
  integer,                          intent(in)    :: from_file, to_file
  integer,                          intent(in)    :: size
  integer(kind=MPI_OFFSET_KIND),    intent(inout) :: file_pos
```

6. Open issues

6.1. Parallel IO performance

Up to now the goal of parallel IO was to make a meaningful parallelisation work, thus its performance is not tuned to the maximum.

When running $(NOUT = 50) * (NT = 100)$ steps and writing the save and dump file the whole simulation takes 960 seconds, of which 62 seconds were used for writing the output. That gives 1.56 seconds per output. For this example all possible dump output was written, which results in a dump file size of 166MB.

This effort is acceptable, but much higher as when using posix files for each processor. In the latter case the same example needs 5.6 seconds of total writing time.

6.2. 64 Bit with MPI1

It is not possible to run CENTORI with 64 Bit and MPICH (MPI 1) on Mare Nostrum. The following error occurs when executing:

```
MPI_ADDRESS : Address of location given to MPI_ADDRESS does not fit in Fortran
     integer
```

This is a known problem. Possible solutions are to use the "MPICH 64-bit Patch" or to migrate to MPI2, of which the latter one is recommended.

6.3. 64 Bit with MPI2

Doesn't work, needs closer examination.

6.4. Using fftw

Using fftw has to be declared in two files. In makefile:

```
FFTW = YES
```

and in centori_control.h by uncommenting the line

```
#define have_fftw
```

But this feature seems not to be mature (many compilation problems) and is recommended to be turned off.

6.5. Uninitialised data

Some data is not initialised when it should be written as output, for example mean square electron density(psi).

6.6. Program crashes

CENTORI crashed in following situations (configurations based on the small test):

- With NRGRID=128 and NZGRID=129 and 512 processors following error occurres for each processor:

  ```
  MXMPI:FATAL-ERROR:0:application called MPI_Abort(MPI_COMM_WORLD, 512) - process 1
  ```

  It works when increasing NRGRID and NZGRID to 257 and 256, respectively.

- Running $(NOUT = 50) * (NT = 100)$ steps on one processor. The error message is

  ```
  slurmd[s03c1b10]: couldn't do a strtol on str 1(1): Numerical result out of range
  slurmd[s03c1b10]: couldn't do a strtol on str 2(2): Numerical result out of range
  [...]
  slurmd[s03c1b10]: couldn't do a strtol on str 14(14): Numerical result out of range
  slurmd[s03c1b10]: couldn't do a strtol on str 515(515): Numerical result out of range
  slurmd[s03c1b10]: couldn't do a strtol on str 531(531): Numerical result out of range
  [...]
  ```

  and so on, in total 27966 lines

A. Configuration files

A.1. Small test

File 1. centori.in for a small test case

```
* CENTORI Global Input File
*** ****************************************************************
***                                                                *
***** MAST standard case                                           *
***   P J Knight 15/06/2010                                        *
***                                                                *
***   Run History:                                                 *
***                                                                *
***   ????: Cold start: ....                                       *
***                                                                *
***                                                                *
*** ****************************************************************
$SIZES
NRGRID      = 128
NZGRID      = 129
RMIN        = 7.5
RMAX        = 200.0
ZMIN        = -200.0
ZMAX        =  200.0
RPMIN       = 7.5
RPMAX       = 145.0
ZPMIN       = -150.0
ZPMAX       =  150.0
$END
$CENTORI
* Max length=50:  12345678901234567890123456789012345678901234567890
RUN_DESCRIPTION = ????:
DUMPFORMAT = 0
DUMP_ALL_ZETA = 0
DT          = 0.5D-9
NX_SPROC    = 1
NY_SPROC    = 1
NZ_SPROC    = 1
NPSI        = 129
NTHETA      = 65
NZETA       = 33
NEQUIL      = 1
NOUT        = 1
NT          = 1000
COLD_START = 1
BREF        = 0.5D4
RREF        = 91.0
PLASMA_CURRENT = 1.0D6
RES_GRASS_INDEX = 2.0
BZERO       = 0.5D4
NEZERO      = 5.0D13
TEZERO      = 1.5
TIZERO      = 2.0
ALPHAN      = 1.6
POWER_E0    = 14.0D6
```

```
ALPHA_POWE = 3.0D0
POWER_I0     = 14.0D6
ALPHA_POWI = 3.0D0
PARTICLE_SRC = 4.0D15
VITOR0        = 0.1
ALPHAVI       = 1.6
SRC_VTOR_MULT = 3.0
DENSITY_FEEDBACK = 1
TARGET_NE   = 3.0D13
TAU_SN        = 1.0D-4
POSITION_CONTROL = 1
RTARGET      = 91.0
CHI_VI         = 0.50D4
CHI_TE         = 0.50D4
CHI_TI         = 0.50D4
CHI_NE         = 1.00D4
CHI_RES        = 0.05D4
RRMULT         = 5.0
JSQ_DMULT    = 4.0
CHI_CLASSICAL_MULT = 0.0
NUPI_MULT    = 0.5
* IONMASS:  1.67D-24 = H,  3.34D-24 = D,  5.01D-24 = T
IONMASS       = 3.34D-24
$END
```