

Installation Manual

ADF Program System

Release 2002.01

SCIENTIFIC COMPUTING & MODELLING NV
Vrije Universiteit; Theoretical Chemistry
De Boelelaan 1083; 1081 HV Amsterdam; The Netherlands
E-mail: support@scm.com



6 March, 2002

COPYRIGHT © 1993–2000: Vrije Universiteit, Theoretical Chemistry, Amsterdam, The Netherlands
All rights reserved.

1	INTRODUCTION	3
1.1	License.....	4
	License file CORRUPT	4
1.2	Platform dependencies.....	6
	Supported Platforms.....	7
2	INSTALLATION INSTRUCTIONS	8
2.1	The ADF package.....	8
	Background Info	8
2.2	External Libraries used by ADF	11
2.2	Environment Variables	12
2.3	Preparation for a Parallel Version	14
	PVM.....	14
	MPI.....	16
2.4	Install Now.....	18
2.5	Customizing your Installation	21
2.6	Test Runs.....	22
	Program execution: run-scripts.....	22
3	INSTALLATION FILES, TECHNICAL INFO	23
3.1	C code in ADF.....	23
3.2	SCU: Parser & Settings	24
3.3	Settings	25
3.4	Makeflags.....	34
3.5	List of environment variables.....	37

1 INTRODUCTION

The Installation Manual describes the installation of the libraries and programs of the ADF package, and the generation of program versions for the computer / operating system (OS) at hand. It also explains how you customize installation parameters that regulate the working size of the program(s) and others that may affect the CPU and I/O performances. The manual is available online at <http://www.scm.com>.

The main programs are ADF and BAND. There are also several utility and property programs. These are used primarily for pre- and post processing data of the main programs. The installation procedure and requirements do not depend on whether or not all of the existing programs and utilities are included in your package. Where in the text the program to be installed is referred it will often simply be denoted ADF. This may, therefore, in fact mean ADF, or BAND, or both, or any of the other programs, as the case may be.

We recommend that you read at least the first parts of this manual. Chapter 1 contains a general discussion of installation issues. Chapter 2 provides a cookbook instruction list for easy installation on platforms that run a UNIX-type OS. Don't worry: the automatic UNIX installation script is straightforward to operate and will usually complete without any difficulties. Nevertheless, it won't hurt to understand how you may improve the installation settings for your situation, and to have some general knowledge of what is going on.

Where references are made to the OS, and more specifically to the file system on your machine, the terminology of a UNIX-type OS will be used. In particular, a hierarchical structure of *directories* is assumed. A non-UNIX OS usually has a similar structure and it should not be too difficult to translate the various concepts from UNIX to your own OS 'language'. There are no fundamental reasons why installing and using ADF in a non-UNIX system would not be possible, but practically speaking we do not really support that.

We strongly recommend a UNIX environment because the development of ADF takes place on UNIX machines and little testing, if at all, is carried out on other platforms. Therefore, although we do the best we can to keep our software as portable as feasible, we cannot guarantee that you won't run into serious trouble on another type of machine. We will offer advise in such cases to see if we can get around any difficulties but again, we can't foresee how that will work out and the human resources that we can allocate to such projects are limited.

The ADF package may be provided partly in source code format and partly as object files, or only as binaries. If you do have access to (parts of) the source codes you have two options: either you install the pre-compiled executables or you compile the sources and combine them with the object files. The automatic (UNIX) installation procedure in the package can be used for either case.

For the second option you need of course a suitable compiler, see the specifications for the supported platforms. For the first option you only need to have the run-time environment. Most likely this is already present on your machine, even if you don't have compilers. Contact us if problems arise with the installation (support@scm.com).

1.1 LICENSE

The programs of the ADF package require a license file to run. However, you may already install the package without the license file. If this is your first installation of ADF we can, in fact, only make your license file *after* you have installed the package.

The license file contains information about the machines that your license allows you to use the programs on. You must make this file available with read-access for all users of the program on all the machines the program might be used on. You can do this, for instance, by putting the license file on a shared disk, or by copying the file to all machines. The name of the license file is immaterial: you have to set an appropriate environment variable to reference it.

We recommend the following procedure:

1. Install the package as described in Chapter 2 (that is: set a couple of environment variables, run our configuration script `configure`, and then run the UNIX `make`).
2. Perform an ‘empty’ test run, by executing the program with an empty input file

```
$ADFBIN/adf << eor  
eor
```

This will fail quickly when the program detects that no valid license file exists.
3. The program will print on output the information that *we* need to create your license file. Send *exactly* that part of the output file to us (email support@scm.com).
4. You will get the license file per e-mail.
5. Pick up the license file and move it to an appropriate location on your machine, corresponding to the environment variable `$SCMLICENSE`, which you should define in your login script (see Chapter 2).
6. Ensure that the permissions on the license file allow read access for all users.
7. Verify that you can now run the sample runs provided with the package and that they give correct results. We recommend that you consult the Examples document for notes on such comparisons: non-negligible differences are not necessarily indicative of an error.

Note that the sample runs are complete run scripts that should be executed as such, not just input files to feed into the program.

License file CORRUPT

You may find that, after having installed the license file, the program still doesn’t run and prints a message like “LICENSE CORRUPT”. There are a few possible causes. To explain how this error may come about, and how you overcome it, a few words on license files.

Each license file consists of pairs of lines. The first of each pair is text that states in more or less readable format (for a human reader) a couple of typical aspects: A ‘feature’ that you are allowed to use (for instance “ADF”), an expiration date, a (maximum) release (version) number of the software and so on. The second line contains the same information in encrypted format: a long string of characters that appear to make little sense. The program reads the license file and checks, with its internal encrypting formulas, that the two lines match. If not, it stops and prints the “corrupt” message. So, there are three common reasons why this may happen to you:

1. You are using a license file for another version of the software than your executables correspond to. Newer (major) releases may contain a different encrypting formula, so that the match in old license files is not recognized anymore. In particular, the 'CORRUPT' error will arise if you run ADF1999 (or later) with a license file that was generated for ADF2.3 (or earlier). Verify that your license file and executable belong to the same major release.
2. The license file as it has been created has been modified in some way. Sometimes, people inspect it and 'clean it up' a little bit, for instance by removing 'redundant spaces', or by making some other 'improvements'. Any such modification will spoil the encryption match and lead to the 'CORRUPT' error. Sometimes the reason lies in the mailing system: if the encrypted line is rather long, the mailer may cut it in two shorter lines. To verify (and correct) this: edit the license file and see if it really consists of pairs of lines as described above. If not, re-unify the broken lines and try again.
You can use the `fixlic` utility to try to fix this automatically. Please be aware that the `fixlic` utility will try to fix the `$SCMLICENSE` file, and replace it with the fixed copy. Thus, you need to make a backup of your license file first, and you need to have write access to the license file `$SCMLICENSE`.

```
cp $SCMLICENSE $SCMLICENSE.backup
$ADFBIN/fixlic
```

3. On some machines (and again, the mailer may play a role here as well) the different lines are separated by two (invisible) characters: `<CR>` and `<NL>` (carriage-return and new-line), while on others (and ours) there's only a `<NL>` to separate the lines. The extra `<CR>` leads to the 'CORRUPT' error. We have included a small utility in our package (`fix_license` in `$ADFHOMe/Install`, which will be copied by the installation script to `$ADFBIN`) to correct this problem. Type

```
fix_license license_file
```

and any `<CR>` characters will be removed from `license_file`. (Your `LANG` environment variable should preferably be set to "C").

1.2 PLATFORM DEPENDENCIES

Unfortunately we can provide some parts of the code only in object-code format, even to those users who have a ‘source-addition’ to their license. These confidential parts contain pieces that we cannot (yet) make available in source code format to ‘the world’. This may apply for example to parts that are still in development and that the authors wish to keep private until their scientific development project has finished. It also applies to software included in the code to control licenses.

Object files are machine dependent and hence we must provide versions of them separately for each of the platforms that we support. This limits in a sense the scope of supported platforms to those where we have generated the appropriate object files. However, if your platform is not included in that list we will, generally speaking, favorably consider a request to generate the required object files if we can get good access to such a machine and suitably modern versions of the operating system and compilers are available.

It is our intention to provide, for each of the supported platforms, both an (almost) complete executable *and* a major part of the package in source code format (for those users who have a license with source codes included). The latter allows you to compile the package yourself, for instance after having modified certain parts to include some of your own developments. Since your compiled results will have to be combined with our pre-compiled object-code parts, your compiler and operating system must match those of the platform where we have generated the object files. The table below lists the supported platforms, with applicable levels of compilers and operating systems. Often (but not always) there is a limited kind of compatibility: it may be possible that objects corresponding to an older version of the compiler can be combined with newer objects. Do not rely on it, however.

The situation is more convenient when it comes to executables only: executables generated under an older level (of compiler and/or operating system) often run without trouble in a newer environment. Also here, however, there is no guarantee and in particular when the OS (and/or compiler) releases are far apart, chances are that you have to upgrade your system. Always test an installed version of our codes carefully with at least a few of the provided sample runs.

There is a large variety among the ADF user sites when it comes to available computing power, memory size, etc., even when the same type of hardware is being used. It is not feasible for us to choose the program size parameters such as maximum numbers of basis functions and atoms in a way that is optimal for everybody. However, barring exceptional circumstances we believe that the defaults that we’ve selected will accommodate virtually all cases. Moreover, you can customize the settings in detail, if necessary, see Chapter 3 “Installation Files, Technical Info”, Section 3.3 “Setting”.

The most important size parameter is the workspace size, which determines (to some extent) how big your calculations can get. The actual choice at compile time is not crucial. It only sets a default, which you can then override in every individual calculation at your discretion. The other size parameters (maximum numbers of basis functions...) have pretty large installation defaults. There’s little chance that you need to increase them and you won’t gain much by decreasing them.

Supported Platforms

<u>Hardware</u>	<u>Operating System</u>	<u>FORTRAN Compiler</u>	<u>C Compiler</u>	<u>Serial/Parallel</u>
Compaq DS20	Digital Unix 4.0	Digital Fortran 90 V5.2-705	DEC C V5.8-009	PVM 3.3.11 (ALPHA)
HP PA2.0	HP-UX 11.01	HP F90 v2.4	HP C Compiler 11.01.03	MPI: HP MPI 1.06
IBM SP	AIX4.3	xlF90 7.1.1	xlC 5.0.0	PVM 3.4.4 (AIX46K)
Pentium	Linux Red Hat 6.2 (Zoot) kernel 2.2.16 #1 smp	Portland Group Compiler pgf90 3.2-3	gcc version egcs- 2.91.66 19990314/Linux	PVM 3.4.2 (LINUX)
SGI Power Challenge	IRIX64 6.2	MIPSpro 7.20		PVM 3.4.3 (SGI6)
SGI Origin3800	IRIX64 6.5	MIPSpro 7.31.2m	MIPSpro 7.31.2m	MPI
NEC Itanium	RedHat 7.1 (NEC)	NEC Fortan Itanium Compiler 6.0, rev 2.1	NEC C++ Itanium Compiler 6.0, rev 2.1	MPI

Generally speaking, a parallel PVM version can also be used as a serial version, for instance when no PVM has been installed at all on your machine and an explicitly serial version is not provided by us. The program will then, in every calculation, print a few warning lines at the very start of the output. These are message from the PVM libraries that have been linked into the executables. They signal that PVM stuff can't be found, which you can simply ignore.

The MPI version may also be used as a serial version (if you don't have an MPI installation on your machine) if the MPI libraries have been linked *statically* in the executable. This may depend on the platform: just try.

A small program version, primarily meaning that the internal workspace size has been kept low, may run in as little space as 32Mb. This will quickly become uncomfortable when you carry out larger calculations: the program will then run less efficiently and may abort with messages indicating that its workspace is too small. In such case you can specify in the input file a larger workspace to use.

The defaults for workspace parameters depend on the platform type. For the workstations and PCs they imply a program size that may increase to over 100 Mb during the calculation, depending on the input. For supercomputers (Cray, NEC, Fujitsu) the default size parameters are some 200 Mb larger. You can allow a bigger (or smaller) extension space on a case by case basis via the input file, see the User manuals.

The main programs (ADF and BAND) may store large amounts of data on disk during a huge calculation. For the ADF molecular program this may run into the hundreds of Mb, for BAND it easily goes into the Gigabyte range. If a parallel version of the program is installed, most of the disk-stored data are spread over the parallel processes, reducing the requirements per node accordingly.

2 INSTALLATION INSTRUCTIONS

If you have a UNIX operating system the installation is easy.

2.1 THE ADF PACKAGE

First, verify that the ADF package has been properly unloaded into your machine. The ADF package comprises:

- (Major parts of) the source codes of the main programs and auxiliary utility programs (the sources are not necessary and may be absent if your license does not include them);
- A few scripts and files that will help you to install the program, such as the parser and different settings files for a variety of platforms.
- A database with basis sets and similar items
- A series of sample calculations: run scripts and output files.

Let `$ADFHOM` be the directory where you've placed the package on your machine. It should (initially) contain the following subdirectories:

`Install`

Files and scripts for the installation procedure;

`atomicdata`

The database with basis sets (and some other materials)

`examples`

Sample runs: run scripts (which contain input files) and output files;

`libtc, libstd`

Source codes to libraries that are shared between different programs;

`commons`

'Include' files: pieces of source code that will be included into various modules of the package at installation / compile time; this is handled by the parser. Typically, the include-files contain Fortran common blocks.

`adf, band, ...`

Source codes to the main programs and to the utility and property programs. Your license determines which programs can be used after the installation has completed; others may not be included at all in your copy of the package.

If you don't have the sources, the directories `libtc, libstd, commons, adf, band, ...` may be empty or even absent.

Background Info

The directory `atomicdata/` contains the database. You need it when you start to do calculations. The directory `examples/` contains sample inputs and outputs. These aspects are discussed in the *User's Guide* and the *Examples* document, respectively.

The source code files, which can be found in the program directories and subdirectories thereof, need to be pre-processed by the parser (SCU) before compilation and are denoted by suffices ".d", ".d90", ".cd". The parser produces files containing plain FORTRAN or C code having a suffix ".f" or ".c". Parsing and compilation are controlled by *make* files used in the installation script.

The directory `Install/` contains a start-up script `configure`, and *make* files that will be addressed by the UNIX *make* command. Apart from a few more files that play little role in the installation, you'll find in `Install/` a number of subdirectories with names like `ibm_xlf3`, `dec_osf1`, and so on: they refer to the platforms that are supported. Each such subdirectory contains a version of the following files:

`lib*.a`

Object files generated for that platform, in archive format and compressed by `gzip`.

`bin*`

Executables generated for that platform, compressed by `gzip`.

`Makeflags`

A file that contains the name of the Fortran and C compiler, compiler flags and the definition of a few 'environment' variables that are used by the UNIX *make* command.

An additional file `Makeflags.mpi` may be present, for the generation of a parallel version that uses MPI as the message passing method. Note that we recommend PVM for reasons of portability in the distribution. The "normal" `Makeflags` file supports both the PVM parallel version and the serial, non-parallel version.

If the platform refers to a non-UNIX situation, no `Makeflags` file will be present at all, since this file is related to the UNIX *make* command.

`settings`

File with parser keys to define platform dependent aspects and program size parameters. We recommend that you do not modify this file. A copy of it, with some adjustments, will be created in the `$ADFHOM` directory later on, which you may then customize to your needs.

If your situation is covered by one of the provided 'platform' directories in `Install` you don't need to carry out any adaptations for the first installation, just apply the right one, as explained below.

The script `Install/configure` is a short UNIX script that creates the files `settings` and `Makeflags` in `$ADFHOM/`, based on the corresponding ones in the appropriate `Install/*` platform subdirectory. It also creates a few 'Makefiles' in `$ADFHOM` and in subdirectories thereof.

`Install/configure` is interactive and performs a dialogue with the user in which various options are explained and choices are presented.

`Install/configure` also creates a number of subdirectories in `$ADFHOM` that are used during the installation process.

Finally it installs the precompiled executables if these are available.

Once everything has been set-up by `Install/configure` you can compile and (re-)install ADF using the UNIX *make* utility in the main directory `$ADFHOM`. You may skip this step if you are content with running the precompiled executables. You have no choice if you don't have the source codes.

When the programs have been installed, the binary executables have been placed in \$ADFBIN, with names like “adf.exe”, “band.exe”, and so on. There will also be files “adf”, “band”. The latter are scripts that execute the binaries. We recommend that you use the script files, rather than the executables directly. When you run a parallel version, the script file will take care of some basic aspects of the parallel run, and it may be inconvenient to do that ‘by hand’ when you call the executable directly. See also the sample run scripts and the Examples document.

2.2 EXTERNAL LIBRARIES USED BY ADF

The programs of the ADF package may for some features require that certain libraries, which are not themselves part of ADF, are installed before you install ADF. These are:

PVM

You can install the PVM version of ADF even if no PVM libraries are available on your machine. However, that implies that the programs will run in serial (non-parallel mode). To use the PVM parallelization, you must have installed PVM (and you must start the PVM daemon before running ADF, see your PVM documentation). PVM is public domain software and can be downloaded from netlib:

<http://www.netlib.org/pvm3/index.html>.

At the moment of this writing the latest version is pvm3.4.3.

GKS, GR

The graphical libraries GKS and GR are required to install and operate the *adfplt* utility program in the ADF package, for graphical display of orbitals, densities and potentials. It is available for free from the authors, at <http://www.KFA-Juelich.de/zam/CompServ/software/graph/soft.html>. You need the GR and GKS libraries. For GKS: go to GLI and select GLI/GKS. These libraries are available (or working correctly) only for a few platforms, unfortunately. We're considering alternatives to provide the same functionality.

2.2 ENVIRONMENT VARIABLES

Before starting the installation script, define the following environment variables, preferably in your *login* script (*csh* syntax is used in this example, with values taken from an actual case). If you use another shell, it may have to go into, say, your *.zshenv* file (for *zsh*) and the syntax may be somewhat different.

```
setenv ADFHOME $HOME/adfhome
```

The directory in which the ADF package has been put.

```
setenv ADFBIN $ADFHOMe/bin
```

The directory in which all executables will be placed. The directory `$ADFBIN` will be created automatically during the installation procedure.

```
setenv ADFRESOURCES $ADFHOMe/atomicdata
```

The directory that contains the database with information for the elemental atoms (basis sets and so on).

```
setenv SCMLICENSE $ADFHOMe/License_File_Name
```

`$SCMLICENSE` specifies where the license file is located. You may not yet have a license file but you can already specify where you will put it later. The value of `$SCMLICENSE` must be the full name (including the path) of the license file.

The next few environment variables play a role in a parallel version. We recommend, however, that you assign appropriate values anyway. They do no harm if you run a serial version.

```
setenv NSCM 16
```

`$NSCM` specifies the *default* number of parallel processes used in a calculation. Its value must be an integer. In each actual calculation this default can be overridden with the `-n` flag to the program-execute scripts. If the environment variable `$NSCM` has not been set, the default number of parallel processes is defined by a specific *key* in the file `$ADFHOMe/settings` (`maxtid`, see the Chapter on Technical Info).

```
setenv TEMPDIR /tmp
```

When running in parallel using PVM, `$TEMPDIR` is the directory where the child processes will create their private (scratch) directories to run in. If possible this directory should be a local directory (i.e., local to the child processes). The master process (the one started in your script) will run in the directory where you start it, it will NOT use `$TEMPDIR`. For the MPI version this variable is currently not used.

```
setenv SCMSPAWNSCRIPT $ADFBIN/adfs
```

`$SCMSPAWNSCRIPT` plays a role only in the parallel PVM version. If it is set at all, it should reference the `adfs` script (using an absolute path) that is put in `$ADFBIN` by the automatic installation procedure. It specifies which script (called from within the program itself) will start up the child processes. If you use a modified version of `adfs`, under a different name, you should adjust the `scmspawnscrip`t variable accordingly.

```
setenv PVM_ARCH SGI6
```

The architecture code for your machine. Consult the PVM documentation for the appropriate name to use.

NOTE: On SGI machines we use for technical reasons the SGI6 architecture. Currently this is required to run ADF or BAND in parallel on the SGI platform using PVM.

```
setenv PVM_ROOT /slocal/pvm3
```

The (master) directory of the PVM installation.

The PVM variables `$PVM_ARCH` and `$PVM_ROOT` have no meaning unless PVM is used. They are not ADF related variables but correspond to the PVM installation on your platform.

The next variable refers to the graphical libraries used by the *adfplt* graphical utility program in ADF. If the variable is not set correctly *adfplt* will not install. If the libraries are not present, you will not be able to use *adfplt*.

```
setenv GKSGR_LIB /usr/local/lib
```

The next environment variable is used (only) in the sample run scripts. It refers to a graphical tool to display contour lines (computed by one of the ADF utility programs) on output.

```
setenv GNUPLOT /slocal/bin/gnuplot
```

The location of the gnuplot graphical program.

2.3 PREPARATION FOR A PARALLEL VERSION

If you want to create a parallel version of ADF using PVM, make sure that PVM has been installed and that the PVM environment variables `$PVM_ROOT` and `$PVM_ARCH` have been set correctly. Consult your PVM documentation.

NOTE: on SGI machines you will need to set `$PVM_ARCH` to `SGI6`.

This has to do with a technical detail within the ADF package that requires the libraries to be compiled using the `-n32` abi on the SGI platforms.

The (main) programs of the ADF package have been parallelized to a large extent. On a parallel machine the computational tasks are distributed over different CPUs. Most of the data are also distributed over the CPUs, storing the data needed by a particular CPU locally as much as possible. The smaller utility programs run in parallel as well, but not all of them have been adapted (yet) to profit significantly from the parallelization.

The management of the processes running on different CPUs and the communication between those processes is rather system dependent. Two different message passing libraries, PVM and MPI, have emerged to date as widespread standards to provide a portable manner to develop parallel programs. MPI appears to become more popular and it is not unlikely that it will soon be the method of preference. Nevertheless, we experience that the way to use MPI to actually run a program is rather system dependent, contrary to PVM. Therefore, PVM is much easier to explain and to prepare for general distribution. This is the (main) reason why we stick to PVM, for the time being, in the general discussion of parallelization. Final note on this subject: although MPI may be more efficient for communication between the processes, this will often be an irrelevant issue in ADF: most of the time is not spent in communication anyway, unless you use a large number of processors. For 2, 4, 8-CPU parallelism, the load balance and other such items dominate the performance issue and these hardly depend on whether you use PVM or MPI.

If possible, install a *parallel* version of the ADF package: you can always run it as a single-node code and it will, apart from minor details, behave as a non-parallel version without any relevant loss of performance.

As already mentioned, our experience so far is that managing the MPI version takes more considerations and more often runs into problems than the PVM version. Therefore, we recommend that you first install PVM (which is public domain software) on your system. Then you can generate the parallel program versions of the ADF suite using PVM for message passing. Most of the remarks and recommendations below about parallel versions refer to the PVM version.

PVM

PVM (Parallel Virtual Machine) provides a common interface for a large number of different parallel architectures, ranging from a heterogeneous cluster of workstations to shared memory multiprocessors. PVM is public domain software and can be obtained for example by `ftp` from `netlib`. Some vendors have their own optimized implementations of PVM to exploit specific hardware features. Unfortunately, these special

versions are not always flawless. Be aware of this when you encounter difficulties and note that you can always install the public domain version instead.

Although MPI seems to become the standard, almost all experience we have with the parallel version of ADF is based on PVM (for historical reasons). This is the main reason why we recommend PVM as the method for message passing for ADF. You can make this choice during the installation process. Information about PVM is available in the PVM User Guide (part of the PVM distribution).

When using PVM, ADF can be executed in the usual way, using the script `$ADFBIN/adf` (which invokes the `adf.exe` binary). This activates the *master* or *parent* process, which will in turn create child processes on the other CPUs in the virtual machine, see the PVM manual.

The start-up of the child ADF processes is controlled by a UNIX script `adfs`, which is called from within the program. The `adfs` script installed automatically during the installation procedure (by `Install/configure`) in `$ADFBIN`.

Note: you may choose another location for the `adfs` script (including the complete path) using the `settings` key `pvmspawnscript`. To do so, edit and adapt the file `$ADFHOMe/settings` *after* `Install/configure` has generated this file but *before* you run `make`. Alternatively you can set this path using the environment variable `$SCMSPAWNSCRIPT` (see chapter 2).

When creating an alternative script, bear in mind that it should

- Create a temporary directory, if possible on a local file system (otherwise the performance may be very poor due to network I/O).
- Make this the new working directory (`'cd'`);
- Call the binary executable in the usual way, but with an empty input file and with standard output redirected to `/dev/null` (“throw away”);
- Clean up the temporary directory.

When the parallel PVM version of ADF, BAND, ... is invoked, a file `nodeinfo` will be searched for and inspected. This file should be located (if it exists at all) in the `$ADFBIN` directory where the executables are generated. If a `nodeinfo` file is found in the current working directory it will be used instead. A PVM version of ADF will spawn processes on all hosts that are known to PVM as being part of the virtual parallel machine (see the PVM manual), but in first instance not more than one process per host would be created. On shared memory systems, like a SGI Origin2000, you may of course want to execute as many processes on a host as there are CPUs in the box. The file `nodeinfo` is used to tell the program how many processes may (at most) be spawned on each of the available hosts. The file must be an ASCII file with format:

```
hostname1  maxproc1
hostname2  maxproc2
(etc.)
```

“maxproc1” et cetera are non-negative integer numbers, and the hostnames should be the names of the hosts available to PVM, as known by PVM. It is not necessary to include all available hosts in the list: for every host not mentioned in `nodeinfo` the program assumes that at most one process should be spawned there. The actual

list of hosts available to PVM can be set and verified easily (use the `conf` command to view the actual list, see the PVM manual). When you have installed PVM, just type

```
pvm
```

This gives you access to a large number of PVM configuration commands. To get a list of available commands, type

```
help
```

To quit from the PVM “console” (but leave PVM running in the background), type

```
quit
```

To terminate PVM, type “halt” instead of “quit”.

The actual total number of parallel processes used in a particular calculation can be set by a command line argument to the program run scripts, for instance

```
adf -n nproc < input
```

`nproc` is the number of processes. The `-n` flag is optional. If it is omitted the default for the number of processes will apply. This is given by the environment variable `$NSCM`, if available, and from the ADF installation / compilation settings otherwise. We recommend to define the `$NSCM` variable in your login settings file, assign it a suitable value and use the `-n` flag to the `start` script in each calculation where you want to apply a different number of parallel processes.

To summarize the meanings and implications of (a) the file `$ADFBIN/nodeinfo`, (b) the environment variable `$NSCM`, (c) the `-n` flag to the program run scripts, and (d) the installation / compilation parameter `maxtid` (see the Chapter with technical information):

- The installation parameter `maxtid` defines an absolute upper bound to the total number of parallel processes in each calculation. It sets, for instance, sizes of arrays in the program and is fixed at compile time. We recommend to give it a safe large value, such as 128 (this is, in fact, the default in the installation procedure). Don't exaggerate, to avoid creating unnecessarily large arrays in the program.
- The file `nodeinfo` defines also an upper bound, specifying in fact the maximum number of parallel processes *per node*. Their sum adds up to the total maximum. Obviously, it is faster and more convenient to change the file `nodeinfo`, than to change `maxtid` and recompile the whole package.
- The actual number of parallel processes used in a calculation is determined by the `-n` flag. If that is absent, it is defined by the `$NSCM` environment variable. If that one doesn't exist either, it is defined by the installation parameter `maxtid`. In all cases, the file `nodeinfo` will limit what is actually used and in no case the total number of processes may exceed `maxtid`.

MPI

MPI is a formal standard message passing facility. We have made it possible for ADF to use MPI instead of PVM for message passing. However, we have only limited experience with the actual usage of the MPI version of ADF and unexpected problems may therefore occur. Details on the installation using MPI are very system dependent, and you may want to consult your MPI User Manual as well as this document to successfully install

the MPI version of ADF. Since many vendors are actively supporting MPI it is quite possible that in the near future MPI will become the method of choice for message passing in the ADF package.

We intend to keep supporting both PVM and MPI as long as they are widespread standards.

2.4 INSTALL NOW...

Run `Install/configure`. This generates all required directories, makefiles, and other installation files (in particular: the two files "Makeflags" and "settings", both in `$ADFHOME`).

`Install/configure` is an interactive script, which should be self-explanatory. If not, let us know!

Follows a typical dialogue with `Install/configure`.

Text printed in ***bold italic*** must be entered by you.

Text printed in *italic* is produced by `Install/configure`.

Normal (non-italic) text is comment / instructions added in this manual for you.

<start of example>

```
cd $ADFHOME
```

```
Install/configure
```

```
Welcome to the configuration script.
```

```
You will be asked some information about the environment in which  
you plan to install and run the programs of the ADF suite.
```

```
-----
```

```
A parallel version can be installed using PVM or MPI
```

```
type pvm      if you want a parallel version using PVM
```

```
type mpi      if you want a parallel version using MPI
```

```
type serial   (or press ENTER) if you want a serial version
```

```
pvm
```

```
Specify the maximum number of processors you want to use.
```

```
Press RETURN for the default ( 128 )
```

```
-----
```

```
On what machine do you install the ADF package?
```

```
(Choose from the following alphabetical list)
```

```
convex
```

```
cray          (C90, J90, T3E)
```

```
dec           ("Alpha")
```

```
fujitsu
```

```
hp           (HP-UX)
```

```
nec          (SX4)
pc_linux
rs6000      (AIX4)
sgi         (IRIX6.2 or higher)
sun4        (SOLARIS 2)
```

rs6000

OK, preparing ...

4967

4967

Creating the PVM version

6335

(more such numbers)

1213

Installing precompiled executables from

/home/tc/visser/adfhome/Install/rs6000_aix/bin.pvm

(done)

To install the programs using your own settings:

type `make (in ADFHOME)`

Before doing so you may edit and modify the files

`ADFHOME/settings` and `ADFHOME/Makeflags`

(The latter contains compiler options).

See the Installation manual for details.

This is the end of the `Install/configure` dialogue. At this point the script has generated the files `settings`, `Makeflags` in `$ADFHOME`.

`settings` contains parser keys, which are described in a next section.

`Makeflags` contains compiler options (and sets a few environment variables to be used by `make`).

Normally you don't need to inspect `settings` and `Makeflags` at all. Of course, it doesn't harm to take a look at these files and try to understand the meaning of the parser keys (in `settings`) and to verify the compile options in `Makeflags`.

When `Install/configure` has finished, the precompiled executables have been installed. If you prefer to generate your own executables use the the UNIX `make` command to compile, link and install the ADF package using your own settings:

`make`

The `make` procedure will compile and link the entire ADF package for you. This may take quite some time, depending on your machine. We've seen cases that took several hours to complete the procedure.

If you have installed the parallel PVM version, you should now create a file `nodeinfo` in the `$ADFBIN` directory in which you specify for all the hosts in the virtual parallel machine how many parallel processes they may run in a calculation. In principle this is the number of CPUs in a host. If this value is one (1) for a particular host, this hostname may be omitted from the list (hosts not found in the file `nodeinfo` but present in the virtual parallel machine are assumed to run at most one process in a calculation).

Make sure that the program run script(s) `adf`, `band`, ... are available for all users by setting the appropriate access permissions.

The directories `$ADFHOMe/OFILES`, and possibly also `$ADFHOMe/GFILES`, contain (archives of) object files of all subroutines and main programs. As soon as the executables have been linked these files are not necessary anymore and you can remove these directories to free some disk space. The same holds for the subdirectories `FFILES`, `MFILES`, `MFILES` and `TSTAMP` in each of the program subdirectories (`$ADFHOMe/adf`, `$ADFHOMe/densf`, `$ADFHOMe/band`, `$ADFHOMe/dirac`, ...): they can be removed when the installation has been completed.

Warning

If, for some reason, an installation fails and you decide to redo it, we recommend that you remove files that could stand in the way in the second attempt by typing (in `$ADFHOMe`):

make clean

Developments in hardware go even faster than in software. New types of platforms are introduced and existing systems are upgraded so often that the installation parameters that are provided in our export package might not be appropriate for your current situation. Similarly, there is such a large number of closely related systems that we cannot possibly provide special installation files for all of them. One should therefore be alert to installation problems, be they a direct failure of the installation procedure, or show up when runs of the examples go wrong. In either case, check the files `$ADFHOMe/Makeflags` and `$ADFHOMe/settings` that you have used. `Makeflags` contains the compiler name and compilation flags, `settings` contains other machine-specifications. If any of these seems to be incorrect, adjust and re-install.

If none of the platform subdirectories in `ADFHOMe/Install/` matches your situation you'll have to make your own version. Take one of the existing platform directories and adjust the files `Makeflags`, `settings` to your needs. The available object files `lib*.a` in the platform-specific subdirectories of `ADFHOMe/Install/` are probably not useful on your machine: remove them and contact us (email: support@scm.com).

2.5 CUSTOMIZING YOUR INSTALLATION

If ADF has already been installed and you want to modify the program, for instance by changing the maximum working size, the *combine* algorithm in case of a parallel version, the maximum number of atoms, et cetera:

Edit and adjust the file `$ADFHOME/settings` that was produced by `Install/configure`. The contents of this file are explained in Chapter 3.

Type

make clean

Type

make

The *clean* step is necessary because the `Makefile` in the ADF installation package is such that `make` will otherwise only check if the source files are out of date. It will not check the date of `$ADFHOME/settings`, or of any of the include files in `$ADFHOME/commons/`. Since a very large number of source files depend on `$ADFHOME/settings` and on the include files, it is not practical to figure out which source files to *touch* so as to let `make` update the program in accordance with the changes to `$ADFHOME/settings` that you made. You better just recompile the entire package. This is effectuated by the *clean* step.

2.6 TEST RUNS

When the installation has completed you should do a couple of test calculations to verify that the programs have been installed correctly. The ADF package comes with a set of sample input and output files. By executing the sample run scripts and comparing the results with the provided output files, several options and features of ADF will be checked. At the same time, doing these test calculations will provide a good start to become familiar with the ins and outs of using the program. Note that the sample run scripts, with names “run.?” are *not* input files, but complete scripts that contain input files *and* UNIX commands.

The User manuals (ADF User’s Guide, BAND User’s Guide, Utilities) explain the general structure of the input files and give an extensive discussion of all available input parameters. The sample runs are briefly discussed in the Examples document.

Program execution: run-scripts

As you can verify by studying a few of the sample run scripts, to execute a program of the ADF package, say *dirac*, you can simply “call” it:

```
$ADFBIN/dirac < in > out
```

“in” is the input file, “out” the standard output file. Depending on the employed program and occasionally on the input file, additional result files may be generated by the program.

The program name, “dirac” in the example, is in fact the name of a script located in your \$ADFBIN directory. It has been put there by the Installation procedure \$ADFHOMe/Install/configure. The program executable itself, which would have the name “dirac.exe” (and which is also located in \$ADFBIN next to the “dirac” run script), is executed by the “dirac” script in \$ADFBIN.

In some cases you could also directly invoke the executable “dirac.exe” itself, but the program run-script automatically takes care of a few issues that play a role when you start using a parallel version. These additional actions don’t affect anything in case of a serial (non-parallel) version. Therefore, we recommend that you use the program run-scripts throughout.

The nice thing of the run scripts is that they have several optional flags, by which you can control details of the execution. To get a survey of the available options, use the “-h” (help) option, by typing:

```
$ADFBIN/dirac -h
```

The scripts have been devised to control details of the parallel execution (as said before) and for debugging purposes. The scripts contribute nothing for daily applications of the programs with a serial version, but they don’t have any negative effect either.

3 INSTALLATION FILES, TECHNICAL INFO

3.1 C CODE IN ADF

ADF uses standard FORTRAN as much as possible to ease porting to different platforms. However, some special facilities have been difficult or impossible to express using standard FORTRAN77, which has for a long time been the development language for our software. A few pieces of C code, which can be found in the directory `libtc/clibs/` if you have source codes, are used within ADF to remedy the deficiencies.

Now that we have started using FORTRAN90 as the basic development language in the ADF package, it is to be expected that most of the C extensions will be removed again: FORTRAN90 provides much of the functionality for which we have inserted the C parts.

C code is currently used in ADF for:

- Interfaces to UNIX routines that return detailed information about resource usage;
- Alternative (on some platforms more efficient) I/O routines (using UNIX I/O calls). These also provide some I/O buffering controlled by the program: data is only written to disk when it cannot be kept in memory anymore.
- Dynamic memory management. Fortran77 did not support dynamic memory allocation.

The basic problem with C, and therefore the main reason to strive for removing it again from package, is not the language itself but the interface between FORTRAN and C, which tends to be rather system dependent. All our C modules include a file `machine.h` which defines the C data types corresponding to FORTRAN data types, and which might add ‘_’ to function names, depending on the machine you are working on. Check this file if there appears to be a problem with the C code and the C-Fortran interfacing and please inform us of any difficulties you encounter!

The file `$ADFHOMe/commons/cexternals.fh` defines a piece of code which is included in each FORTRAN routine that invokes C routines. In case of problems with the FORTRAN – C interface you may want to check this file.

The C extensions are not essential for the proper functioning of ADF (although they may play a role in convenience and performance). Most relevant is probably the suppression of truly dynamic memory management if you turn the C extensions off. Suppressing all C code is achieved by adjusting the appropriate key in `$ADFHOMe/settings`. Just set:

```
ccode=false
```

Note that you must adjust `$ADFHOMe/settings` *after* you have run the `Install/configure` script to generate that file in the first place.

IF you have set `ccode=false` in `settings`, you must also edit and adjust the file `$ADFHOMe/Makeflags`, which contains a line specifying the parameter `CCODE`: set it to `false`. We recommend, however, that you do not change the ‘ccode’ setting. Consult us if you believe that the C extensions cause problems in your ADF installation.

3.2 SCU: PARSER & SETTINGS

To handle installation aspects for different environments comfortably we employ a set-up called SCU (Source Code Utility). SCU allows an easy and detailed definition of the program version you want to create without the need to check the complete source code for relevant statements and declarations. This flexibility translates in a number of parameters, or *keys* that you must assign appropriate values.

For a large number of platforms we provide pre-set values of these keys – and some other items – that should be usable without modification in ordinary circumstance. For other platforms, and when particular situations require different values, the user may want to change them. This is an easy task and takes nothing more than editing and adjusting one or two small ASCII files at an early phase of the installation procedure. All involved issues are described in this manual.

The ADF system comes with several so-called `settings` files, in which all the installation keys have been given suitable values for a number of different platforms, and for a normal program size. Chapter 3 ‘The Settings File’ explains how you can adapt the settings, for instance if you want to port the program to a platform that is not represented by one of the existing `settings` files. The `settings` file also defines the program size: the amount of workspace internally available (by default), the maximum number of atoms, etc.

The SCU setup implies simply that the source code files are pre-processed by a *parser*. The parser reads the `settings` and generates the appropriate FORTRAN (or C, as the case may be) source files, which are then compiled. The parser program is contained in the package and is operated automatically by the UNIX installation procedure.

The program source files are stored in SCU format: FORTRAN code intermixed with directives for the parser. The directives tell the parser to insert additional code and/or to choose between alternative pieces of code. The parser reads the SCU file, interprets the directives and generates a FORTRAN source which can then be compiled. With the SCU keys in `ADFHOME/settings` you customize ADF to the operating system and other requirements. Versions for different platforms, or with different characteristics for the same platform, are constructed by adapting the SCU parameters and letting the parser generate the corresponding FORTRAN (and C) source.

For more information about the SCU setup and how include-files, Fortran and C procedures and modules are stored in the ADF package and handled by the installation script, see also the Programming manual (“rules”) of the ADF package.

3.3 SETTINGS

All files in the ADF package with the name `settings` consist of a sequence of *keys* with associated values. We will discuss here only the most important of these files, which is the one in the main directory of the package: `$ADFHOME/settings`.

The file starts with a record

```
*key
```

and it ends with a record

```
*keyend
```

empty records have no meaning and have only been inserted for the human reader. The same holds for records starting with ``!'``: these are just comments.

All key records have the form

```
key=value
```

where `value` is everything after the equal sign: it may be just a number or a word, or more general text including any number of embedded spaces.

Each key is defined by only one line; the value-part must not extend to the next line.

The keys in `$ADFHOME/settings` characterize to a large extent the platform and the program size. Follows an enumeration of all keys in `$ADFHOME/settings`, with an explanation of their meaning. Key values shown have been taken from an actual case (installation on a SGI 4 CPU Power Challenge).

```
platform=sgi_irix64
```

Characterization of the platform. The name matches one of the platform subdirectories in `$ADFHOME/Install`. This key may be used in the code to determine platform dependent code alternatives.

```
pvmroot=/usr/local/pvm3
```

The absolute path to the `$PVM_ROOT` directory. This is the place where the PVM library and 'include' files can be found. When PVM has been installed and configured properly, this key should have the same value as the environment variable `$PVM_ROOT`.

```
pvmarch=SGI6
```

The PVM machine architecture name: `$PVM_ARCH`. It indicates the architecture of your computer. The appropriate values are specified in the PVM documentation. When PVM has been properly installed and configured this key should match the environment variable `$PVM_ARCH`.

The key values for `pvmroot`, `pvmarch` have been written into the `settings` file by the `Install/configure` script using the environment variables `$PVM_ROOT` and `$PVM_ARCH`, which should have been set correctly in advance.

```
pvm=true
```

flags whether of not a parallel version is installed that uses PVM as the message passing library.

```
mpi=false
```

flags whether of not a parallel version is installed that uses MPI as the message passing library.

Note that the `pvm` and `mpi` keys must not both be true. However, they may be both *false*: in that case a *serial* version is installed. The values of the `pvm` and `mpi` keys are written into the `settings` file by the `configure` script, in which you have specified in the dialogue what kind of program version you wish to install.

`fpvm3_h=/usr/local/pvm3/include/fpvm3.h`

the absolute path to the `fpvm3.h` 'include' file. This file is part of the `pvm3` distribution and contains a number of constants used by any PVM program. The automatically supplied value in `settings` refers to the usual location of this value. You should verify that it matches your situation.

`maxtid=128`

the maximum number of processes used in the parallel execution. Normally this is the same as the number of CPUs in your parallel system. This value does not imply that each run parallelizes to this extent, it is just a maximum that defines array sizes in the source code. Displayed in this example is the default value 128.

`parallel=true`

specifies whether or not a parallel version is being generated. In the `settings` file you will see that its value is not specified as true or false directly, but as true *if either one of the keys `pvm`, `mpi` is true*.

`release=99.Mar`

ADF release number

`releasedate=25 March, 1999`

Date of release.

Release number and date are printed in output files and in standard result files. The release number may also be checked at some places in the code. Do not change them, as they are part of copy right and proprietary labels and statements.

`broadcastoption=1`

Option to determine the algorithm used to broadcast data from the parent to the child processes. A list of possibilities is given below. If an option has been implemented such that it can only be used for a particular message passing method (PVM, MPI, ...) such is indicated between parentheses:

- 1: use a binary tree algorithm (PVM)
- 2: use `pvmfmcast` from the master to all kids (PVM)
- 3: use `pvmfmcast` from the master to everyone (including the master) (PVM)
- 4: use `mpi_bcast` (MPI)

`gatheroption=5`

Determines which algorithm will be used to gather distributed vectors and matrices to form complete matrices.

The `gatheroption` is one of the following:

- 1: use a wild algorithm: everyone sends everything to everybody else (PVM)
- 2: use a ring algorithm, matrices are broken in many messages (PVM)
- 3: use a binary tree algorithm, followed by a broadcast (PVM)
- 4: use `mpi_allgather` (MPI)
- 5: use a ring algorithm, matrices in one piece (PVM)

`combineoption=1`

Determines which algorithm will be used to combine (sum) the values from all processes and distribute the resulting value to everyone. This is also called a global reduce operation. The `combineoption` is one of the following:

- 1: use a binary tree combineto the master, followed by a broadcast (PVM)
- 2: use a binary tree combine/scatter in one (PVM)

3: use a partial combine to all followed by broadcast (PVM)

4: use `mpi_allreduce` (MPI)

5: use `pvmfreduce` and broadcast (PVM)

`rcv_timeout`

In a parallel execution the processes will wait for expected messages no longer than this amount of time (seconds); after this period the program will consider whether the communication is a problem (maybe another process doesn't live anymore); see also the next key.

`max_rcv_timeouts`

Maximum number of successive attempts to receive the expected message within the `rcv_timeout` period of time, see the previous key. After the last such attempt has failed the program will abort execution assuming that at least one of the parallel processes doesn't live anymore.

`haspvmreduce=true`

Informs the program whether the PVM version used has a global *reduce* routine called *pvmfreduce*. If it has, it might be used depending on the *combineoption*. Some vendor-specific implementations of PVM (for instance PVM-e) do not have *pvmfreduce*.

`mpi_h=mpif.h`

Path to the `mpi.h` 'include' file. This file is part of the MPI distribution and contains a number of constants used by any MPI program. Its value is irrelevant if a parallel version is generated based on PVM rather than MPI.

`pvmspawnscript=/home/local/scm/adfhome/bin/adfs`

File name of an executable UNIX script, or the ADF executable itself, including a full path.

When using PVM, the master ADF executable creates the child processes by starting this script on the other CPUs. The script, which normally should be located in the `$ADFBIN` directory, should create a new, unique directory, `cd` to it, and start the program copy of the child process, with *no* input file and standard output redirected to `/dev/null`. The directory will be used for scratch files and must be in the usual local scratch space (system dependent, something like `/tmp`). The default value for this key is `$ADFBIN/adfs`, and an example `adfs` script is created in `$ADFBIN` by the `Install/configure` procedure.

Take a look at `$ADFBIN/adfs` if you consider to adapt it to your situation. Consult also the PVM documentation.

`ccode=true`

Indicates whether C code in the package may be used. The value set by `Install/configure` should normally not be changed, except in special circumstances, for instance when it was set *true*, but you don't have a C compiler. If the value is *false*, you must make sure that the variable `CCODE` in the file `adfhome/Makeflags` is also *false*.

`pvmargs32=false`

For platforms with 8-byte integers this must be true to ensure that calls to PVM procedures are adjusted internally so as to supply 4-byte integers to PVM. The value for this parameter should have been set correctly by the `Install/configure` script.

`pvmroutedirect=true`

This optimizes communication (PVM version), assuming that the PVM version you have installed supports the direct communication between processes, rather than via the `pvm` daemon.

`hasowndirectory=true`

This must be *false* only if different parallel processes may use the same directory to generate their files in. This may be the case on shared-memory systems and on some other systems where you can *not* use a UNIX

script to start the child processes. In these cases the files generated by the parent and child process would have the same name, leading to conflict. ADF remedies this internally by appending an integer (preceded by an underscore) to the file names (if *hasowndirectory* is *false*). As a result, during the calculation there will then be files TAPE21_0, TAPE21_1, et cetera instead of TAPE21. At the end of the calculation, however, the script renames the files belonging to the parent process (the _0 files) by removing the _0 suffix, so for the user there's no difference (after the calculation has finished) with the situation where each process has its own directory.

`pvmspawnntasks=true`

Flags whether the *pvmspawnscript* is used to spawn the child processes. On some systems this will (automatically) be done in another way and the parent process must then not try to do so from within the program.

`pvmtragicevents=true`

Flags whether or not the PVM package used supports sending information about 'tragic events' (host down, child process killed, ...) to the parallel processes.

According to our experience so far, all platforms that we support have PVM versions that do support this, except the Cray systems.

`timing=true`

If true, a resource usage analysis of the entire program is internally generated and optionally (see the User's Guide) reported on output. This key should normally be true because the involved computational overhead is small; see the next key, however.

`manytimers=false`

If true, a very large number of timing checks are generated. This may slow down your calculation by over 30% and is used for performance debugging. In fact, the number of timer calls under this flag may vary wildly from one release to another, since it is in fact a development tool.

`timingbarrier=false`

If true, a parallel run will at certain points in the code let all processes wait for the slowest one, so as to make sure that all processes start at the same moment on the next section. This is used for performance analysis.

Turning it on should not have a major impact on performance, but you better leave it off.

`debug=true`

This is an almost-obsolescent key. It turns on a few (i.e. hardly any) checks in the code. They have little practical meaning, but at the other hand they also are insignificant for the performance.

`scalapack=false`

If *true*, the SCALAPACK is used for diagonalization. Currently this option is not (yet) supported and the SCALAPACK code is not included. This key must therefore be *false*.

`adinputfile=adinput`

Sets the path to an ADF input file.

This key tells ADF to try to read from this file *if it exists* instead of using standard input. The reason for this option is that some non-UNIX systems do not provide the usual stdin/stdout redirect facilities. In such case the user should copy the input to the `adinputfile`.

`bytes/real=4`

Describes the number of bytes taken by a *single-precision* real on the platform at hand. **DO NOT** adjust this to 8, with the idea that you want to use a double-precision version of the program. The effect of your modification would, on the contrary, be that the program 'thinks' that single-precision reals are already 8 bytes and that double-precision arithmetic is not necessary.

Allowed values for are: 4, 8, 16. If your machine happens to have, for instance, 60 bits for a real, the corresponding value for the key would be 7.5. You must then put 8, being one of the allowed values that most closely matches the case.

`doubleimag=DIMAG`

AIMAG is the standard function. If a specific function is available for complex*16 arguments you should supply that name here. For instance, DIMAG is the name of this function on the IBM rs6000 stations running AIX.

`doubleconjug=DCONJG`

The same remarks as for the previous key apply here.

`doublecplx=DCMPLX`

Similarly. The function can have one or two (real or double precision) arguments and yields a complex, respectively complex*16 result.

`bytes/integer=4`

As for the key `bytes/real`: this specifies how single-precision (default) integers are defined. Some machines have 2 bytes per integer. We are not sure whether or not this will cause problems. Contact us in such case.

`bytes/integer32=4`

To scale down memory used by integers, some (minor) parts of the code use special integer declarations.

These should, normally, be 4 bytes as reflected by this key.

`integer32=INTEGER(KINT)`

Declaration name for the above-meant integers. If the “integer32” variables and default integers are identical, the declaration should also be the same. Default integers in ADF are declared as `INTEGER(KINT)`. `KINT` is a `KIND` parameter that is defined in the program.

`logfileunit=3`

Fortran file unit number used for the `logfile`. Should only be changed if the chosen value is not available (reserved for standard error, for instance).

`licenceunit=4`

Similar considerations: Fortran channel used for reading the license file from within the program.

`standardinput=5`

Fortran unit of standard input.

`standardoutput=6`

Fortran unit of standard output.

`logfile=logfile`

The programs of the ADF package write messages (strings) to a log file. This ASCII file can be inspected during the run so as to check what the program is currently doing, whether there are important warnings, and so on. It is a concise summary of the run. Upon normal termination the program will copy the logfile to the end of the standard output file, which contains much more data.

`filepathcharacter=/`

The value must be the character that separates the (local) file name from the path description. In UNIX systems this is the slash (/).

`filename=160`

The maximum length, in characters, of a file name (including its path). This applies to the Fortran `OPEN` statement:

`open (iu=unit, file=filename,)`

This key is becoming obsolescent and is not consistently used in the program. Don't change it.

`filedescriptors=200`

Maximum number of file descriptors used within the C-based I/O software within the ADF package.

`fileunits=99`

The maximum number of Fortran channels possibly used by the program. The value must (currently) still be rather large for BAND. For ADF_molecular it plays a smaller role.

`fliounits=65`

The maximum number of Fortran channels used by a package of modules within BAND. It plays no role for ADF. This number must be smaller than the value for *fileunits* above.

`reclengthunit=1`

In the Fortran77 `OPEN` statement for direct-access files the record length must be specified. The standard is not clear about this. *reclengthunit* is the number of bytes of the basic unit in which the parameter is defined. Example: on IBM rs/6000 workstations *reclengthunit* must be 1 (the parameter defines the recordlength in bytes), while on the SGI PowerChallenge it must be 4 (the parameter defines the recordlength in single-precision reals, each being 4 bytes).

`eofscan=true`

This key flags whether Fortran read statements with the `END=` parameter are executed correctly. Although this should of course be the case, the construction being perfectly standard Fortran77, not all machines do this as they should.

`memadressunit=1`

Used for the alignment of memory blocks

`nopointer=false`

Flags whether the platform supports the usage of pointers within FORTRAN77. With the introduction of Fortran90, which does support pointers formally, this will become obsolescent. However, not all parts of the code have been transferred to Fortran90 yet.

`dynamicmemory=true`

Flags usage of truly dynamic memory management (using C code) from the Fortran77 program parts, rather than pseudo-dynamic allocations of arrays (using standard Fortran77) inside a large 'work' array in labelled common.

`dynamicreals=$dynamicmemory`

By default linked to the value of the `dynamicmemory` key.

`dynamicintegers=$dynamicmemory`

See previous, now for integer type data

`dynamiclogicals=$dynamicmemory`

See previous, for logicals.

`dynamicstrings=$dynamicmemory`

See previous, for character data.

`realmemblock=1`

Default size of 'chunks' of memory that are added to / removed from the workspace that the program uses internally to allocate arrays in of real (or double precision) data type. Specification is in Mbytes and the value may be non-integer, see below for an example. When no dynamic memory is used, this is the size of the *total amount* of memory available for reals (using a Fortran `COMMON` block).

`integermemblock=1/2`

Similar for integers

logicalmemblock=1/32

... and for logicals

stringmemblock=3/2

... and for strings.

maxmemoryusage=48

This key plays an important role. It defines the default maximum amount of 'scratch' memory (Mbytes) available during a calculation. If *dynamicmemory* is *true*, its value can be overridden via the input file for each actual calculation. If *dynamicmemory* is *false*, however, it has no meaning and the amount of memory available depends only on *realmemblock*, *integermemblock* and so on.

When *dynamicmemory* is on, the previous keys relating to memory management are merely technical and you should generally not touch them. This one, however, more or less defines your program-size together with the *iobuffersize* key.

Final note: if *dynamicmemory* is false, the 'amount of memory' that you set in the configure dialogue will be assigned (in the settings file) to the *realmemblock* variable, which is then the amount of memory used for reals. This is therefore not the total amount of memory for scratch space since integers, logicals and strings are not included. These together take a little more than 4 Mb in addition.

iobuffersize=4

Specifies in Mb how much memory is used as a buffer space for IO operations. Data to be stored on disk (as 'defined' by 'write' calls to the KF library routines) are actually kept in the IO buffer in memory. On machines with slow IO (over network) this may significantly improve the IO performance. On other systems it may just duplicate what the operating system already provides for.

atomtypes=50

Maximum number of different *types* of atoms that can be used in one and the same calculation. Although this is formally not the same as the number of chemical elements, it will in many practical cases coincide with it.

See the User's Guide for more comments on 'atom type' in ADF.

atoms=1000

Maximum number of atoms allowed in a calculation.

atomicorbitals=1000

Maximum number of basis function *types*, defined by a basis function and an atom *type*. Compare the key *basisfunctions*.

basisfunctions=20000

Maximum number of basis functions in a calculation. The difference with the *atomicorbitals* key is that if, for instance, you use 30 Carbon atoms, each having say 12 basis functions centered on them, the corresponding value (or contribution to) *atomicorbitals* would be 12, while the value for *basisfunctions* would be $30 \times 12 = 360$.

coresets=1000

Maximum number of expansion function characteristics for the frozen core orbitals. This does not count the expansion over different atoms of an atom type, as for *atomicorbitals*, nor does it count the expansion over the distinct *m*-values for a particular angular momentum quantum number *l*.

fitsets=2000

Maximum number of expansion function characteristics (as for *coresets*) for the fit functions that are used to solve Poisson's equation by an approximate expansion of the true charge density in one-center functions.

vectors=64

Default (maximum) vector length. Should be fair, and reasonably large on a typical vector machine. Excessive values offer little advantage and only negatively affect memory usage and cache usage. The value actually used in a calculation will depend on the available memory and the size of the system to be computed. On non-vector machines, experience suggests that 64 is a good value.

```
expolimit=500.0_KREAL
```

The cut-off value to prevent underflow. $\text{EXP}(-A)$ will be set to zero when A exceeds *expolimit*. Note that in the definition a Fortran90 KIND specification is included. The KREAL parameter is defined within the program to be the standard real declaration (corresponding to the Fortran77 double precision on 4-byte real machines, for instance).

```
powerlimit=(0.4343_KREAL * $expolimit)
```

This prevents underflow for expressions like $10.0^{*(-A)}$.

```
directscf=true
```

Controls the default method (which be overridden via input for any particular calculation) to process large amounts of data: recompute or store-on-disk-and-retrieve.

```
cio=true
```

Controls whether C code is used for the majority of IO operations. It improves performance on some systems and doesn't seem to harm on other platforms. On some we have, by default, disabled it for technical reasons.

We recommend that you do not set it *true* when the standard value is *false* for your system.

```
skipfitints=true
```

Enables suppressing the storage of almost-zero integrals on disk that are used in the 'fit' procedure. This issue is still under development and has no practical meaning yet.

```
kfpackedreals=true
```

Enables the compression of data when writing to disk. Should improve the IO performance somewhat.

```
datatypes=5
```

Defines the size of arrays that run over different data types used in certain IO processing modules. Don't change this value. The 5 distinct data types are integer, logical, real, string and complex; each of these has one of the values 1...5 to code for it, see the next keys.

```
integertype=1
```

See previous.

```
logicaltype=2
```

See prvioius

```
realtype=3
```

See previous

```
stringtype=4
```

See previous

```
complexttype=5
```

See previous

```
! -----  
! most of the remaining keys are standard and/or controlled by  
! the parser, the user should not change them at installation  
! -----
```

The remainder is omitted: not relevant for the installation. Note, however, at the end of the settings file the part that starts with “`if not ccode`”, where consistency between some earlier defined keys is enforced.

3.4 MAKEFLAGS

The file `Makeflags` contains a few environment variables that you must give appropriate values. You may also want to modify this file to use better compiler-optimization flags than those provided in the standard distribution. Please notify us if you find that the standard settings can be improved *without invoking unsafe options*.

Following is an example (taken from a SGI installation) with comments.

```
FCPARS=f90
```

The Fortran compiler used to compile the parser (pre-processor).

```
FFPARS=-O2 -n32 -mips3 -align64 -nocpp -wofall -G 0
```

Fortran options used in compiling the parser.

```
LFPARS=-O2 -n32 -mips3 -align64 -nocpp -woffall -G 0
```

Link flags used for linking the parser

```
CC=cc
```

The C compiler used for compiling C modules in ADF

```
CFLAGS=-n32 -mips32 -O2 -Dsgi_irix64
```

Flags used for the C compiler

```
CDBFLAGS=...
```

Flags used for the C compilations for the debug version of the programs. This is used only in development environments.

```
MOD=mod
```

Suffix for Fortran90 module files.

```
LINKMOD=yes
```

flags that compiled Fortran90 modules should be linked.

```
FC=f90 -n32 - mips3 -align64 -nocpp -OPT:Olimit=0 -G 0 -woffall
```

Name of the Fortran compiler used for compiling the ADF sources, with any set of flags that are to be used in the compilation of each subroutine or function.

```
O0V0=-O1
```

Fortran flags for routines that have shown to be problematic for optimizers *and* for which vectorization is potentially counter-effective for the performance. You may assume that the routines compiled with this flag are insignificant for the overall performance of the programs.

Note that any flags included in the “FC” definition, see above, will also be used for all routines compiled with the O0V1 flags. The same holds for the next few sets of flags.

```
O1V0=-O2
```

Fortran flags: moderate optimization, no vectorization.

```
O2V0=-O3
```

```
-OPT:IEEE_arithmetic=1:got_call_conversion=OFF:swp=ON:pad_common=OFF: (...)
```

Fortran flags: for maximum optimization, but potentially no vectorization.

```
O0V1, O1V1, O2V1
```

as for the previous 3 keys, but now with vectorization included.

```
MODFLAGS
```

Compiler flags for compiling Fortran90 modules. These should (in the current version) not be performance critical, so a moderate optimization is just fine. Some of the Fortran90 modules are rather complex, so high optimizations may run into compiler bugs (present-day Fortran90 compilers are found to be unstable / buggy, in some cases).

```
MSEARCH=-I ../$(PROGDIR)/MFILES -I ../libtc/MFILES
```

Flags to specify where compiled Fortran90 module files are to be found.

Note: the way Fortran90 modules are handled differs quite a bit from one system to another. The related flags (MODFLAGS, LINKMOD, MSEARCH) may therefore be rather different between different platforms.

```
LFLAGS=-O2 -n32 -mips3 -align64 -G 0 -woffall
```

Flags used for the debug version. This is used only in development environments.

```
DBMODFLAGS
```

included in the link statement.

```
DBFLAGS=...
```

Fortran, MGSEARCH, LDBFLAGS are the counterparts for the debug version of the 'normal' flags

MODFLAGS, MSEARCH, LFLAGS

```
AR=ar
```

The standard archive command. If it is not available in your standard search path, or if it has a different name, you have to specify the correct name here, including the full path if necessary.

```
RANLIB=echo Finished
```

This example turns RANLIB off and replaces it by a simple echo of "Finished". For other platforms you may need to include a ranlib link flag here.

```
LIBEXTRA=
```

Additional libraries (...) that you may want to link. Be careful with standard Lapack libraries: we have a slightly modified version included in the ADF package and had some difficulties with the standard installation on certain platforms. In addition, BLAS and LAPACK are not heavily used in ADF, so using highly optimized pre-installed libraries is not likely to improve the performance much.

```
CCODE=true
```

This value must be adjusted if the key *ccode* in *adfhomesettings* is false!!

```
PVMUSE=-L/usr/local/pvm3/lib/RS6K -lfpvm3 -lpvm3 -lfpvm3
```

PVM libraries to be linked into the executables (for a PVM version, of course)

```
platform=sgi_irix64
```

Name and characterization of the platform; used in the make script.

```
ADFHOME=/home/local/scm/adfhomes
```

Directory where the package is being installed

```
ADFBIN=/home/local/scm/adfhomes/bin
```

Directory where exes are put by the installation script

```
INSTALL=$(ADFHOME)/Install
```

Directory with Installation information files

```
OFILES=$(ADFHOME)/OFILES
```

Directory to put object files during the installation procedure

```
GFILES=$(ADFHOME)/GFILES
```

Same, but now for the object files of the debug version.

The six keys '*OnVm*' define compiler flags used for compiling the normal ADF source codes. The six different sets of such flags are applied respectively to different functions and subroutines that, for instance, should *not* be optimized or vectorized. Most of the routines, and in particular the most time-consuming ones will be compiled with the flags given for the 'high-optimize, vectorize' variety. Therefore, if you consider to experiment with improving performance, adapt only the *O2VI* key. The other flag sets are mainly there to prevent catastrophes due to known compiler bugs, and to suppress vectorization that would very probably lead to a significant *loss* of performance.

The items *ADFHOME*, *ADFBIN*, *INSTALL*, *OFILES*, *GFILES* are written into the `Makeflags` file by the `Install/configure` script. Their values should normally be OK.

3.5 LIST OF ENVIRONMENT VARIABLES

Following is an alphabetically ordered list of environment variables used by the ADF package. Typical values (not default values) have are shown next to the variable, and a short description is given. For most situations it will not be necessary to change (or set) any of these environment variables.

ADFBIN \$ADFHOMe/bin

Absolute path to the directory containing the binaries and scripts for the ADF package.

ADFHOMe \$HOMe

Absolute path to the directory in which the ADF distribution has been installed.

ADFRRESOURCES \$ADFHOMe/atomicdata

Absolute path to the atomicdata directory.

DUPHOST (either set or unset)

Do not limit the number of tasks to one per CPU (PVM only).

NSCM 4

Number of tasks to run (evidently applies to the parallel vesion only).

PVM_ARCH SGI6

PVM architecture identifier (see PVM documentation).

PVM_ROOT \$HOMe/pvm3

Absolute path to directory containing your pvm3 distribution (see PVM documentation).

SCMSPAWNSCRIPT \$ADFBIN/adfs

Absolute path to a script which will generate the child processes (started by PVM).

TEMPDIR /tmp/\$USER

Absolute path to a directory in which the child processes during a PVM run will execute.

SCMBCOP 1

Broadcast algorithm to use (overrides value of broadcastoption in settings file).

SCMCBOP 1

Combine (global sum) algorithm to use (overrides value of combineoption in settings file).

SCMGAOP 5

Gather algorithm to use (overrides value of gatheroption in settings file).

SCMLICENSE \$ADFHOMe/license.adf

Absolute path to the location of the license file, has precedence over SCMLICENSE.

SCMLICENSE \$ADFHOMe/license.adf

Absolute path to the location of the license file, if SCMLICENSE has not been set.

SCM_MAX_RCV_TIMEOUTS 2

Number of times to try to receive a message before halting the program (overrides value of max_rcv_timeouts in settings file).

SCM_RCV_TIMEOUT 900

Number of seconds to wait for a message before halting the program (but try SCM_MAX_RCV_TIMEOUTS times first) (overrides value of rcv_timeout in settings file).

SCM_VECTORLENGTH 1024

The block length to use (number of integration point handled at the same time), see the description of the vectors key. If set, this value overrides the value in the settings file.

SCM_IOBUFFERSIZE 100

The amount of memory to use for buffering IO to KF-files, identical to the iobuffersize settings key. If set, it overrides the value specified in the settings file.