User's Guide for CP (version 5.0)

Contents

1 Introduction **1**

2 Compilation **2**

3 Input data **2**
3.1 Data files . . . 3
3.2 Format of arrays containing charge density, potential, etc. . . . 3

4 Using CP **4**
4.1 Reaching the electronic ground state . . . 6
4.2 Relax the system . . . 6
4.3 CP dynamics . . . 9
4.4 Advanced usage . . . 10
4.4.1 Self-interaction Correction . . . 10
4.4.2 ensemble-DFT . . . 12
4.4.3 Free-energy surface calculations . . . 14
4.4.4 Treatment of USPPs . . . 14

5 Performances **14**

1 Introduction

This guide covers the usage of the CP package, version 5.0, a core component of the QUANTUM ESPRESSO distribution. Further documentation, beyond what is provided in this guide, can be found in the directory CPV/Doc/, containing a copy of this guide.

This guide assumes that you know the physics that CP describes and the methods it implements. It also assumes that you have already installed, or know how to install, QUANTUM ESPRESSO. If not, please read the general User's Guide for QUANTUM ESPRESSO, found in directory Doc/ two levels above the one containing this guide; or consult the web site: http://www.quantum-espresso.org.

People who want to modify or contribute to CP should read the Developer Manual: Doc/developer_man.pdf.

CP can perform Car-Parrinello molecular dynamics, including variable-cell dynamics, and free-energy surface calculation at fixed cell through meta-dynamics, if patched with PLUMED.

The CP package is based on the original code written by Roberto Car and Michele Parrinello. CP was developed by Alfredo Pasquarello (IRRMA, Lausanne), Kari Laasonen (Oulu), Andrea Trave, Roberto Car (Princeton), Nicola Marzari (Univ. Oxford), Paolo Giannozzi, and others. FPMD, later merged with CP, was developed by Carlo Cavazzoni, Gerardo Ballabio (CINECA), Sandro Scandolo (ICTP), Guido Chiarotti (SISSA), Paolo Focher, and others. We quote in particular:

- Manu Sharma (Princeton) and Yudong Wu (Princeton) for maximally localized Wannier functions and dynamics with Wannier functions;
- Paolo Umari (Univ. Padua) for finite electric fields and conjugate gradients;
- Paolo Umari and Ismaila Dabo for ensemble-DFT;
- Xiaofei Wang (Princeton) for META-GGA;
- The Autopilot feature was implemented by Targacept, Inc.

This guide has been mostly writen by Gerardo Ballabio and Carlo Cavazzoni.

We shall greatly appreciate if scientific work done using this code will contain an explicit acknowledgment and the following reference:

> P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. Dal Corso, S. Fabris, G. Fratesi, S. de Gironcoli, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. P. Seitsonen, A. Smogunov, P. Umari, R. M. Wentzcovitch, J.Phys.:Condens.Matter 21, 395502 (2009), http://arxiv.org/abs/0906.2569

2 Compilation

CP is included in the core QUANTUM ESPRESSO distribution. Instruction on how to install it can be found in the general documentation (User's Guide) for QUANTUM ESPRESSO.

Typing make cp from the main QUANTUM ESPRESSO directory or make from the CPV/ subdirectory produces the following codes in CPV/src:

- cp.x: Car-Parrinello Molecular Dynamics code
- cppp.x: postprocessing code for cp.x
- wfdd.x: utility code for finding maximally localized Wannier functions using damped dynamics.

Symlinks to executable programs will be placed in the `bin/` subdirectory.

As a final check that compilation was successful, you may want to run some or all of the tests and examples. Please see the general User's Guide for their setup. Automated tests for `cp.x` are in directory `tests/` and can be run via the script `check_cp.x.j`

You may take the tests and examples distributed with `CP` as templates for writing your own input files. Input files for tests are contained in `tests/` subdirectory with file type `*.in1`, `*.in2`, ... Input file for examples are produced if you run the examples in the `results/` subdirectories, with names ending with `.in`.

For general information on parallelism and how to run in parallel execution, please see the general User's Guide. `CP` currently can take advantage of both MPI and OpenMP parallelization. The "plane-wave", "linear-algebra" and "task-group" parallelization levels are implemented.

3 Input data

Input data for `cp.x` is organized into several namelists, followed by other fields introduced by keywords. The namelists are

&CONTROL:	general variables controlling the run
&SYSTEM:	structural information on the system under investigation
&ELECTRONS:	electronic variables, electron dynamics
&IONS :	ionic variables, ionic dynamics
&CELL (optional):	variable-cell dynamics

The `&CELL` namelist may be omitted for fixed-cell calculations. This depends on the value of variable `calculation` in namelist &CONTROL. Most variables in namelists have default values. Only the following variables in &SYSTEM must always be specified:

`ibrav`	(integer)	Bravais-lattice index
`celldm`	(real, dimension 6)	crystallographic constants
`nat`	(integer)	number of atoms in the unit cell
`ntyp`	(integer)	number of types of atoms in the unit cell
`ecutwfc`	(real)	kinetic energy cutoff (Ry) for wavefunctions.

).

Explanations for the meaning of variables `ibrav` and `celldm`, as well as on alternative ways to input structural data, are contained in files `Doc/INPUT_CP.*`. These files are the reference for input data and describe a large number of other variables as well. Almost all variables have default values, which may or may not fit your needs.

After the namelists, you have several fields ("cards") introduced by keywords with self-explanatory names:

ATOMIC_SPECIES
ATOMIC_POSITIONS
CELL_PARAMETERS (optional)
OCCUPATIONS (optional)

The keywords may be followed on the same line by an option. Unknown fields are ignored. See the files mentioned above for details on the available "cards".

3.1 Data files

The output data files are written in the directory specified by variable `outdir`, with names specified by variable `prefix` (a string that is prepended to all file names, whose default value is: `prefix='pwscf'`). The `iotk` toolkit is used to write the file in a XML format, whose definition can be found in the Developer Manual. In order to use the data directory on a different machine, you need to convert the binary files to formatted and back, using the `bin/iotk` script.

The execution stops if you create a file `prefix.EXIT` in the working directory. NOTA BENE: this is the directory where the program is executed, NOT the directory `outdir` defined in input, where files are written. Note that with some versions of MPI, the working directory is the directory where the `pw.x` executable is! The advantage of this procedure is that all files are properly closed, whereas just killing the process may leave data and output files in unusable state.

3.2 Format of arrays containing charge density, potential, etc.

The index of arrays used to store functions defined on 3D meshes is actually a shorthand for three indices, following the FORTRAN convention ("leftmost index runs faster"). An example will explain this better. Suppose you have a 3D array `psi(nr1x,nr2x,nr3x)`. FORTRAN compilers store this array sequentially in the computer RAM in the following way:

```
        psi(   1,   1,   1)
        psi(   2,   1,   1)
        ...
        psi(nr1x,   1,   1)
        psi(   1,   2,   1)
        psi(   2,   2,   1)
        ...
        psi(nr1x,   2,   1)
        ...
        ...
        psi(nr1x,nr2x,   1)
        ...
        psi(nr1x,nr2x,nr3x)
etc
```

Let `ind` be the position of the `(i,j,k)` element in the above list: the following relation

```
        ind = i + (j - 1) * nr1x + (k - 1) *  nr2x * nr1x
```

holds. This should clarify the relation between 1D and 3D indexing. In real space, the `(i,j,k)` point of the FFT grid with dimensions `nr1` ($\leq$`nr1x`), `nr2` ($\leq$`nr2x`), , `nr3` ($\leq$`nr3x`), is

$$r_{ijk} = \frac{i-1}{nr1}\tau_1 + \frac{j-1}{nr2}\tau_2 + \frac{k-1}{nr3}\tau_3$$

where the τ_i are the basis vectors of the Bravais lattice. The latter are stored row-wise in the `at` array: τ_1 = `at(:, 1)`, τ_2 = `at(:, 2)`, τ_3 = `at(:, 3)`.

The distinction between the dimensions of the FFT grid, `(nr1,nr2,nr3)` and the physical dimensions of the array, `(nr1x,nr2x,nr3x)` is done only because it is computationally convenient in some cases that the two sets are not the same. In particular, it is often convenient to have `nrx1`=`nr1`+1 to reduce memory conflicts.

4 Using CP

It is important to understand that a CP simulation is a sequence of different runs, some of them used to "prepare" the initial state of the system, and other performed to collect statistics, or to modify the state of the system itself, i.e. modify the temperature or the pressure.

To prepare and run a CP simulation you should first of all define the system:

atomic positions
system cell
pseudopotentials
cut-offs
number of electrons and bands (optional)
FFT grids (optional)

An example of input file (Benzene Molecule):

```
&control
   title = 'Benzene Molecule',
   calculation = 'cp',
   restart_mode = 'from_scratch',
   ndr = 51,
   ndw = 51,
   nstep = 100,
   iprint = 10,
   isave = 100,
   tstress = .TRUE.,
   tprnfor = .TRUE.,
   dt    = 5.0d0,
   etot_conv_thr = 1.d-9,
   ekin_conv_thr = 1.d-4,
   prefix = 'c6h6',
   pseudo_dir='/scratch/benzene/',
   outdir='/scratch/benzene/Out/'
/
&system
   ibrav = 14,
   celldm(1) = 16.0,
   celldm(2) = 1.0,
   celldm(3) = 0.5,
   celldm(4) = 0.0,
   celldm(5) = 0.0,
   celldm(6) = 0.0,
   nat = 12,
   ntyp = 2,
   nbnd = 15,
   ecutwfc = 40.0,
   nr1b= 10, nr2b = 10, nr3b = 10,
   input_dft = 'BLYP'
```

```
/
&electrons
   emass = 400.d0,
   emass_cutoff = 2.5d0,
   electron_dynamics = 'sd'
/
&ions
   ion_dynamics = 'none'
/
&cell
   cell_dynamics = 'none',
   press = 0.0d0,
 /
 ATOMIC_SPECIES
 C 12.0d0 c_blyp_gia.pp
 H 1.00d0 h.ps
 ATOMIC_POSITIONS (bohr)
 C     2.6 0.0 0.0
 C     1.3 -1.3 0.0
 C    -1.3 -1.3 0.0
 C    -2.6 0.0 0.0
 C    -1.3 1.3 0.0
 C     1.3 1.3 0.0
 H     4.4 0.0 0.0
 H     2.2 -2.2 0.0
 H    -2.2 -2.2 0.0
 H    -4.4 0.0 0.0
 H    -2.2 2.2 0.0
 H     2.2 2.2 0.0
```

You can find the description of the input variables in file `Doc/INPUT_CP.*`.

4.1 Reaching the electronic ground state

The first run, when starting from scratch, is always an electronic minimization, with fixed ions and cell, to bring the electronic system on the ground state (GS) relative to the starting atomic configuration. This step is conceptually very similar to self-consistency in a `pw.x` run.

Sometimes a single run is not enough to reach the GS. In this case, you need to re-run the electronic minimization stage. Use the input of the first run, changing `restart_mode = 'from_scratch'` to `restart_mode = 'restart'`.

NOTA BENE: Unless you are already experienced with the system you are studying or with the internals of the code, you will usually need to tune some input parameters, like `emass`, `dt`, and cut-offs. For this purpose, a few trial runs could be useful: you can perform short minimizations (say, 10 steps) changing and adjusting these parameters to fit your needs. You can specify the degree of convergence with these two thresholds:

`etot_conv_thr`: total energy difference between two consecutive steps
`ekin_conv_thr`: value of the fictitious kinetic energy of the electrons.

Usually we consider the system on the GS when `ekin_conv_thr` $< 10^{-5}$. You could check the value of the fictitious kinetic energy on the standard output (column EKINC).

Different strategies are available to minimize electrons, but the most used ones are:

- steepest descent: `electron_dynamics = 'sd'`
- damped dynamics: `electron_dynamics = 'damp'`, `electron_damping` = a number typically ranging from 0.1 and 0.5

See the input description to compute the optimal damping factor.

4.2 Relax the system

Once your system is in the GS, depending on how you have prepared the starting atomic configuration:

1. if you have set the atomic positions "by hand" and/or from a classical code, check the forces on atoms, and if they are large ($\sim 0.1 \div 1.0$ atomic units), you should perform an ionic minimization, otherwise the system could break up during the dynamics.
2. if you have taken the positions from a previous run or a previous ab-initio simulation, check the forces, and if they are too small ($\sim 10^{-4}$ atomic units), this means that atoms are already in equilibrium positions and, even if left free, they will not move. Then you need to randomize positions a little bit (see below).

Let us consider case 1). There are different strategies to relax the system, but the most used are again steepest-descent or damped-dynamics for ions and electrons. You could also mix electronic and ionic minimization scheme freely, i.e. ions in steepest-descent and electron in with damped-dynamics or vice versa.

(a) suppose we want to perform steepest-descent for ions. Then we should specify the following section for ions:

```
&ions
  ion_dynamics = 'sd'
/
```

Change also the ionic masses to accelerate the minimization:

```
ATOMIC_SPECIES
 C 2.0d0 c_blyp_gia.pp
 H 2.00d0 h.ps
```

while leaving other input parameters unchanged. *Note* that if the forces are really high (> 1.0 atomic units), you should always use steepest descent for the first (~ 100 relaxation steps.

(b) As the system approaches the equilibrium positions, the steepest descent scheme slows down, so is better to switch to damped dynamics:

```
&ions
  ion_dynamics = 'damp',
  ion_damping = 0.2,
  ion_velocities = 'zero'
/
```

A value of `ion_damping` around 0.05 is good for many systems. It is also better to specify to restart with zero ionic and electronic velocities, since we have changed the masses.

Change further the ionic masses to accelerate the minimization:

```
ATOMIC_SPECIES
C 0.1d0 c_blyp_gia.pp
H 0.1d0 h.ps
```

(c) when the system is really close to the equilibrium, the damped dynamics slow down too, especially because, since we are moving electron and ions together, the ionic forces are not properly correct, then it is often better to perform a ionic step every N electronic steps, or to move ions only when electron are in their GS (within the chosen threshold).

This can be specified by adding, in the ionic section, the `ion_nstepe` parameter, then the &IONS namelist become as follows:

```
&ions
  ion_dynamics = 'damp',
  ion_damping = 0.2,
  ion_velocities = 'zero',
  ion_nstepe = 10
/
```

Then we specify in the &CONTROL namelist:

```
etot_conv_thr = 1.d-6,
ekin_conv_thr = 1.d-5,
forc_conv_thr = 1.d-3
```

As a result, the code checks every 10 electronic steps whether the electronic system satisfies the two thresholds `etot_conv_thr`, `ekin_conv_thr`: if it does, the ions are advanced by one step. The process thus continues until the forces become smaller than `forc_conv_thr`.

Note that to fully relax the system you need many runs, and different strategies, that you should mix and change in order to speed-up the convergence. The process is not automatic, but is strongly based on experience, and trial and error.

Remember also that the convergence to the equilibrium positions depends on the energy threshold for the electronic GS, in fact correct forces (required to move ions toward the minimum) are obtained only when electrons are in their GS. Then a small threshold on forces could not be satisfied, if you do not require an even smaller threshold on total energy.

Let us now move to case 2: randomization of positions.

If you have relaxed the system or if the starting system is already in the equilibrium positions, then you need to displace ions from the equilibrium positions, otherwise they will not move in a dynamics simulation. After the randomization you should bring electrons on the GS again, in order to start a dynamic with the correct forces and with electrons in the GS. Then you should switch off the ionic dynamics and activate the randomization for each species, specifying the amplitude of the randomization itself. This could be done with the following &IONS namelist:

```
&ions
  ion_dynamics = 'none',
  tranp(1) = .TRUE.,
  tranp(2) = .TRUE.,
  amprp(1) = 0.01
  amprp(2) = 0.01
/
```

In this way a random displacement (of max 0.01 a.u.) is added to atoms of species 1 and 2. All other input parameters could remain the same. Note that the difference in the total energy (etot) between relaxed and randomized positions can be used to estimate the temperature that will be reached by the system. In fact, starting with zero ionic velocities, all the difference is potential energy, but in a dynamics simulation, the energy will be equipartitioned between kinetic and potential, then to estimate the temperature take the difference in energy (de), convert it in Kelvin, divide for the number of atoms and multiply by 2/3. Randomization could be useful also while we are relaxing the system, especially when we suspect that the ions are in a local minimum or in an energy plateau.

4.3 CP dynamics

At this point after having minimized the electrons, and with ions displaced from their equilibrium positions, we are ready to start a CP dynamics. We need to specify `'verlet'` both in ionic and electronic dynamics. The threshold in control input section will be ignored, like any parameter related to minimization strategy. The first time we perform a CP run after a minimization, it is always better to put velocities equal to zero, unless we have velocities, from a previous simulation, to specify in the input file. Restore the proper masses for the ions. In this way we will sample the microcanonical ensemble. The input section changes as follow:

```
&electrons
   emass = 400.d0,
   emass_cutoff = 2.5d0,
   electron_dynamics = 'verlet',
   electron_velocities = 'zero'
/
&ions
   ion_dynamics = 'verlet',
   ion_velocities = 'zero'
/
ATOMIC_SPECIES
```

```
C 12.0d0 c_blyp_gia.pp
H 1.00d0 h.ps
```

If you want to specify the initial velocities for ions, you have to set `ion_velocities ='from_input'`, and add the IONIC_VELOCITIES card, after the ATOMIC_POSITION card, with the list of velocities in atomic units.

NOTA BENE: in restarting the dynamics after the first CP run, remember to remove or comment the velocities parameters:

```
&electrons
   emass = 400.d0,
   emass_cutoff = 2.5d0,
   electron_dynamics = 'verlet'
   ! electron_velocities = 'zero'
/
&ions
   ion_dynamics = 'verlet'
   ! ion_velocities = 'zero'
/
```

otherwise you will quench the system interrupting the sampling of the microcanonical ensemble.

Varying the temperature It is possible to change the temperature of the system or to sample the canonical ensemble fixing the average temperature, this is done using the Nosé thermostat. To activate this thermostat for ions you have to specify in namelist &IONS:

```
&ions
   ion_dynamics = 'verlet',
   ion_temperature = 'nose',
   fnosep = 60.0,
   tempw = 300.0
/
```

where `fnosep` is the frequency of the thermostat in THz, that should be chosen to be comparable with the center of the vibrational spectrum of the system, in order to excite as many vibrational modes as possible. `tempw` is the desired average temperature in Kelvin.

Note: to avoid a strong coupling between the Nosé thermostat and the system, proceed step by step. Don't switch on the thermostat from a completely relaxed configuration: adding a random displacement is strongly recommended. Check which is the average temperature via a few steps of a microcanonical simulation. Don't increase the temperature too much. Finally switch on the thermostat. In the case of molecular system, different modes have to be thermalized: it is better to use a chain of thermostat or equivalently running different simulations with different frequencies.

Nośe thermostat for electrons It is possible to specify also the thermostat for the electrons. This is usually activated in metals or in systems where we have a transfer of energy between ionic and electronic degrees of freedom. Beware: the usage of electronic thermostats is quite delicate. The following information comes from K. Kudin:

"The main issue is that there is usually some "natural" fictitious kinetic energy that electrons gain from the ionic motion ("drag"). One could easily quantify how much of the fictitious energy comes from this drag by doing a CP run, then a couple of CG (same as BO) steps, and then going back to CP. The fictitious electronic energy at the last CP restart will be purely due to the drag effect."

"The thermostat on electrons will either try to overexcite the otherwise "cold" electrons, or it will try to take them down to an unnaturally cold state where their fictitious kinetic energy is even below what would be just due pure drag. Neither of this is good."

"I think the only workable regime with an electronic thermostat is a mild overexcitation of the electrons, however, to do this one will need to know rather precisely what is the fictitious kinetic energy due to the drag."

4.4 Advanced usage

4.4.1 Self-interaction Correction

The self-interaction correction (SIC) included in the `CP` package is based on the Constrained Local-Spin-Density approach proposed my F. Mauri and coworkers (M. D'Avezac et al. PRB 71, 205210 (2005)). It was used for the first time in QUANTUM ESPRESSO by F. Baletto, C. Cavazzoni and S.Scandolo (PRL 95, 176801 (2005)).

This approach is a simple and nice way to treat ONE, and only one, excess charge. It is moreover necessary to check a priori that the spin-up and spin-down eigenvalues are not too different, for the corresponding neutral system, working in the Local-Spin-Density Approximation (setting `nspin = 2`). If these two conditions are satisfied and you are interest in charged systems, you can apply the SIC. This approach is a on-the-fly method to correct the self-interaction with the excess charge with itself.

Briefly, both the Hartree and the XC part have been corrected to avoid the interaction of the excess charge with tself.

For example, for the Boron atoms, where we have an even number of electrons (valence electrons = 3), the parameters for working with the SIC are:

```
&system
nbnd= 2,
total_magnetization=1,
sic_alpha = 1.d0,
sic_epsilon = 1.0d0,
sic = 'sic_mac',
force_pairing = .true.,

&ions
ion_dynamics = 'none',
ion_radius(1) = 0.8d0,
sic_rloc = 1.0,

ATOMIC_POSITIONS (bohr)
B 0.00 0.00 0.00 0 0 0 1
```

The two main parameters are:

`force_pairing = .true.`, which forces the paired electrons to be the same;
`sic='sic_mac'`, which instructs the code to use Mauri's correction.

Remember to add an extra-column in ATOMIC_POSITIONS with "1" to activate SIC for those atoms.

Warning: This approach has known problems for dissociation mechanism driven by excess electrons.

Comment 1: Two parameters, `sic_alpha` and `sic_epsilon'`, have been introduced following the suggestion of M. Sprik (ICR(05)) to treat the radical (OH)-H_2O. In any case, a complete ab-initio approach is followed using `sic_alpha=1`, `sic_epsilon=1`.

Comment 2: When you apply this SIC scheme to a molecule or to an atom, which are neutral, remember to add the correction to the energy level as proposed by Landau: in a neutral system, subtracting the self-interaction, the unpaired electron feels a charged system, even if using a compensating positive background. For a cubic box, the correction term due to the Madelung energy is approx. given by $1.4186/L_{box} - 1.047/(L_{box})^3$, where L_{box} is the linear dimension of your box (=celldm(1)). The Madelung coefficient is taken from I. Dabo et al. PRB 77, 115139 (2007). (info by F. Baletto, francesca.baletto@kcl.ac.uk)

4.4.2 ensemble-DFT

The ensemble-DFT (eDFT) is a robust method to simulate the metals in the framework of "ab-initio" molecular dynamics. It was introduced in 1997 by Marzari et al.

The specific subroutines for the eDFT are in `CPV/src/ensemble_dft.f90` where you define all the quantities of interest. The subroutine `CPV/src/inner_loop_cold.f90` called by `cg_sub.f90`, control the inner loop, and so the minimization of the free energy A with respect to the occupation matrix.

To select a eDFT calculations, the user has to set:

```
calculation = 'cp'
occupations= 'ensemble'
tcg = .true.
passop= 0.3
maxiter = 250
```

to use the CG procedure. In the eDFT it is also the outer loop, where the energy is minimized with respect to the wavefunction keeping fixed the occupation matrix. While the specific parameters for the inner loop. Since eDFT was born to treat metals, keep in mind that we want to describe the broadening of the occupations around the Fermi energy. Below the new parameters in the electrons list, are listed.

- `smearing`: used to select the occupation distribution; there are two options: Fermi-Dirac smearing='fd', cold-smearing smearing='cs' (recommended)
- `degauss`: is the electronic temperature; it controls the broadening of the occupation numbers around the Fermi energy.
- `ninner`: is the number of iterative cycles in the inner loop, done to minimize the free energy A with respect the occupation numbers. The typical range is 2-8.
- `conv_thr`: is the threshold value to stop the search of the 'minimum' free energy.

- `niter_cold_restart`: controls the frequency at which a full iterative inner cycle is done. It is in the range 1÷`ninner`. It is a trick to speed up the calculation.

- `lambda_cold`: is the length step along the search line for the best value for A, when the iterative cycle is not performed. The value is close to 0.03, smaller for large and complicated metallic systems.

NOTE: `degauss` is in Hartree, while in `PWscf`is in Ry (!!!). The typical range is 0.01-0.02 Ha.
The input for an Al surface is:

```
&CONTROL
 calculation = 'cp',
 restart_mode = 'from_scratch',
 nstep  = 10,
 iprint = 5,
 isave  = 5,
 dt    = 125.0d0,
 prefix = 'Aluminum_surface',
 pseudo_dir = '~/UPF/',
 outdir = '/scratch/'
 ndr=50
 ndw=51
/
&SYSTEM
 ibrav=  14,
 celldm(1)= 21.694d0, celldm(2)= 1.00D0, celldm(3)= 2.121D0,
 celldm(4)= 0.0d0,   celldm(5)= 0.0d0, celldm(6)= 0.0d0,
 nat= 96,
 ntyp= 1,
 nspin=1,
 ecutwfc= 15,
 nbnd=160,
 input_dft = 'pbe'
 occupations= 'ensemble',
 smearing='cs',
 degauss=0.018,
/
&ELECTRONS
 orthogonalization = 'Gram-Schmidt',
 startingwfc = 'random',
 ampre = 0.02,
 tcg = .true.,
 passop= 0.3,
 maxiter = 250,
 emass_cutoff = 3.00,
 conv_thr=1.d-6
 n_inner = 2,
 lambda_cold = 0.03,
```

```
 niter_cold_restart = 2,
/
&IONS
 ion_dynamics  = 'verlet',
 ion_temperature = 'nose'
 fnosep = 4.0d0,
 tempw = 500.d0
/
ATOMIC_SPECIES
 Al 26.89 Al.pbe.UPF
```

NOTA1 remember that the time step is to integrate the ionic dynamics, so you can choose something in the range of 1-5 fs.
NOTA2 with eDFT you are simulating metals or systems for which the occupation number is also fractional, so the number of band, `nbnd`, has to be chosen such as to have some empty states. As a rule of thumb, start with an initial occupation number of about 1.6-1.8 (the more bands you consider, the more the calculation is accurate, but it also takes longer. The CPU time scales almost linearly with the number of bands.)
NOTA3 the parameter `emass_cutoff` is used in the preconditioning and it has a completely different meaning with respect to plain CP. It ranges between 4 and 7.

All the other parameters have the same meaning in the usual `CP` input, and they are discussed above.

4.4.3 Free-energy surface calculations

Once `CP` is patched with `PLUMED` plug-in, it becomes possible to turn-on most of the PLUMED functionalities running `CP` as: `./cp.x -plumed` plus the other usual `CP` arguments. The PLUMED input file has to be located in the specified `outdir` with the fixed name `plumed.dat`.

4.4.4 Treatment of USPPs

The cutoff `ecutrho` defines the resolution on the real space FFT mesh (as expressed by `nr1`, `nr2` and `nr3`, that the code left on its own sets automatically). In the USPP case we refer to this mesh as the "hard" mesh, since it is denser than the smooth mesh that is needed to represent the square of the non-norm-conserving wavefunctions.

On this "hard", fine-spaced mesh, you need to determine the size of the cube that will encompass the largest of the augmentation charges - this is what `nr1b`, `nr2b`, `nr3b` are. hey are independent of the system size, but dependent on the size of the augmentation charge (an atomic property that doesn't vary that much for different systems) and on the real-space resolution needed by augmentation charges (rule of thumb: `ecutrho` is between 6 and 12 times `ecutwfc`).

The small boxes should be set as small as possible, but large enough to contain the core of the largest element in your system. The formula for estimating the box size is quite simple:

$$\texttt{nr1b} = 2R_c/L_x \times \texttt{nr1}$$

and the like, where R_{cut} is largest cut-off radius among the various atom types present in the system, L_x is the physical length of your box along the x axis. You have to round your result

to the nearest larger integer. In practice, `nr1b` etc. are often in the region of 20-24-28; testing seems again a necessity.

The core charge is in principle finite only at the core region (as defined by some R_{rcut}) and vanishes out side the core. Numerically the charge is represented in a Fourier series which may give rise to small charge oscillations outside the core and even to negative charge density, but only if the cut-off is too low. Having these small boxes removes the charge oscillations problem (at least outside the box) and also offers some numerical advantages in going to higher cut-offs." (info by Nicola Marzari)

5 Performances

`cp.x` can run in principle on any number of processors. The effectiveness of parallelization is ultimately judged by the "scaling", i.e. how the time needed to perform a job scales with the number of processors, and depends upon:

- the size and type of the system under study;
- the judicious choice of the various levels of parallelization (detailed in Sec.**??**);
- the availability of fast interprocess communications (or lack of it).

Ideally one would like to have linear scaling, i.e. $T \sim T_0/N_p$ for N_p processors, where T_0 is the estimated time for serial execution. In addition, one would like to have linear scaling of the RAM per processor: $O_N \sim O_0/N_p$, so that large-memory systems fit into the RAM of each processor.

As a general rule, image parallelization:

- may give good scaling, but the slowest image will determine the overall performances ("load balancing" may be a problem);
- requires very little communications (suitable for ethernet communications);
- does not reduce the required memory per processor (unsuitable for large-memory jobs).

Parallelization on k-points:

- guarantees (almost) linear scaling if the number of k-points is a multiple of the number of pools;
- requires little communications (suitable for ethernet communications);
- does not reduce the required memory per processor (unsuitable for large-memory jobs).

Parallelization on PWs:

- yields good to very good scaling, especially if the number of processors in a pool is a divisor of N_3 and N_{r3} (the dimensions along the z-axis of the FFT grids, `nr3` and `nr3s`, which coincide for NCPPs);
- requires heavy communications (suitable for Gigabit ethernet up to 4, 8 CPUs at most, specialized communication hardware needed for 8 or more processors);

- yields almost linear reduction of memory per processor with the number of processors in the pool.

A note on scaling: optimal serial performances are achieved when the data are as much as possible kept into the cache. As a side effect, PW parallelization may yield superlinear (better than linear) scaling, thanks to the increase in serial speed coming from the reduction of data size (making it easier for the machine to keep data in the cache).

VERY IMPORTANT: For each system there is an optimal range of number of processors on which to run the job. A too large number of processors will yield performance degradation. If the size of pools is especially delicate: N_p should not exceed N_3 and N_{r3}, and should ideally be no larger than $1/2 \div 1/4 N_3$ and/or N_{r3}. In order to increase scalability, it is often convenient to further subdivide a pool of processors into "task groups". When the number of processors exceeds the number of FFT planes, data can be redistributed to "task groups" so that each group can process several wavefunctions at the same time.

The optimal number of processors for "linear-algebra" parallelization, taking care of multiplication and diagonalization of $M \times M$ matrices, should be determined by observing the performances of `cdiagh/rdiagh` (`pw.x`) or `ortho` (`cp.x`) for different numbers of processors in the linear-algebra group (must be a square integer).

Actual parallel performances will also depend on the available software (MPI libraries) and on the available communication hardware. For PC clusters, OpenMPI (`http://www.openmpi.org/`) seems to yield better performances than other implementations (info by Kostantin Kudin). Note however that you need a decent communication hardware (at least Gigabit ethernet) in order to have acceptable performances with PW parallelization. Do not expect good scaling with cheap hardware: PW calculations are by no means an "embarrassing parallel" problem.

Also note that multiprocessor motherboards for Intel Pentium CPUs typically have just one memory bus for all processors. This dramatically slows down any code doing massive access to memory (as most codes in the QUANTUM ESPRESSO distribution do) that runs on processors of the same motherboard.