Goals of parallel computing

Typical goals of (non-trivial) parallel computing in electronic-structure calculations:

- To speed up calculations that would take too much time on a single processor. A good **scaling** is required, i.e.: execution time should ideally scale linearly, $T_N \propto T/N$, with number N of processors. *Amdahl's law: if a fraction* P *of the code is parallelized, the maximum speedup is* $S_N = 1/(1 - P + P/N)$, *where* $T_N = T/S_N$

- To make it possible to perform calculations that would never fit into a single processor RAM. Requires **memory distribution**: the total required RAM should be distributed across processors to prevent *memory bottlenecks*

Important concepts in parallel computing

In order to achieve a good scaling, one should pay attention to

- **Load Balancing**: in the ideal parallel code, all processors should perform roughly the same amount of computation

- **Communication**: in the ideal parallel code, communication between processors (to exchange data) should take a small amount of the total time – communication is much slower than computation!

Parallelization of electronic-structure codes based on plane waves and pseudopotentials (PW-PP) is far from trivial.

Parallel programming paradigms

The main parallelization paradigms are MPI and OpenMP:

- **MPI (Message Passing Interface) parallelization**
  Many processes are executed in parallel, one per processor, accessing their own set of variables. Access to variables residing on another processor is achieved via explicit calls to MPI libraries.

- **OpenMP parallelization**
  A single process spawns sub-processes (*threads*) on other processors that can access and process the variables of the code. Achieved with compiler directives and/or via call to *multi-threading* libraries like Intel MKL or IBM ESSL.

Quantum ESPRESSO exploits both MPI and OpenMP parallelization.

Comparison of parallel programming paradigms

MPI parallelization:

+ *Portable* to all architectures (as long as MPI libraries exist)

+ Can be very *efficient* (as long as communication hardware is, and provided MPI libraries make good use of it)

+ Can scale up to a *large* number of processors

– Requires *significant* code reorganization and rewriting

Comparison of parallel programming paradigms

OpenMP parallelization:

+ Relatively *easy* to implement: requires modest code reorganization and rewriting – similar to "vectorization" in old CRAY vector machines

– *Not* very efficient: doesn't scale well on more than a few processors. Interesting for multi-core CPUs and, in combination with MPI, for large parallel machines based on multi-core CPUs. Typical example: IBM BlueGene class machines.

Relevant variables in PW-PP electronic-structure codes that determine the computational workload

N_w:	number of plane waves (used in wavefunction expansion)
N_g:	number of **G**-vectors (used in charge density expansion)
N_1, N_2, N_3:	dimensions of the FFT grid for charge density (for Ultrasoft PPs there are two distinct grids)
N_a:	number of atoms in the unit cell or supercell
N_e:	number of electron (Kohn-Sham) states (bands)
N_p:	number of projectors in nonlocal PPs (sum over cell)
N_k:	number of **k**-points in the irreducible Brillouin Zone

Relevant variables in PW-PP codes (2)

Note that

$$N_w \propto V \quad \text{(size of the unit cell)},$$

$$N_w \propto E_c^{3/2} \quad \text{(kinetic energy cutoff)}$$

and

$$N_g \sim N_r = N_1 \times N_2 \times N_3,$$

$$N_g \sim \alpha N_w$$

with $\alpha = 8$ for norm-conserving, $\alpha \sim 20$ for ultrasoft PP;

$$N_p \sim \beta N_a, \quad N_e \sim \gamma N_a \quad (\beta, \gamma \sim 10), \quad N_a << N_w.$$

Computational effort $\propto N_a^2 \div N_a^3$, RAM $\propto N_a^2$.

Time-consuming operations

- Calculation of density, $n(\mathbf{r}) = \sum |\psi(\mathbf{r})|^2$ (+ augmentation terms for USPP): FFT + linear algebra (matrix-matrix multiplication)

- Calculation of potential, $V(\mathbf{r}) = V_{xc}[n(r)] + V_H[n(\mathbf{r})]$: FFT + operations on real-space grid

- Iterative diagonalization (SCF) / electronic force (CP) calculation, $H\psi$ products: FFT + linear algebra (matrix-matrix multiplication)

- Subspace diagonalization (SCF) / Iterative orthonormalization of Kohn-Sham states (CP): diagonalization of $N_e \times N_e$ matrices + matrix-matrix multiplication

Basically: most CPU time spent in linear-algebra operations, implemented in BLAS and LAPACK libraries, and in FFT

Memory-consuming arrays

- **Wavefunctions** (Kohn-Sham orbitals):
  at least n_k arrays (at least 2 for CP) of size $N_w \times N_e$, plus a few more copies used as work space. Arrays can be stored to disk and used one at the time: less memory, more disk I/O

- **Charge density and potential**:
  several vectors of size N_g (in **G**-space) or N_r (in **R**-space) each

- **Pseudopotential projectors**:
  a $N_w \times N_p$ array containing $\beta_i(\mathbf{G})$

- **Miscellaneous Matrices**:
  $N_e \times N_e$ arrays containing $\langle\psi_i|\psi_j\rangle$ products used in subspace diagonalization or iterative orthonormalization; $N_e \times N_p$ arrays containing $\langle\psi_i|\beta_j\rangle$ products

Required actions for effective parallelization of PW-PP electronic-structure codes

- **Balance load**:
  all processors should have the same load as much as possible
- **Reduce communications to the strict minimum**:
  communication is typically much slower than computation!
- **Distribute all memory that grows with the size of the unit cell**: if you don't, you will run out of memory for large cells
- **Distribute all computation that grows with the size of the unit cell**:
  if you don't, sooner or later non-parallelized calculations will take most of the time (a practical demonstration of Amdhal's law)

Implementation solutions

The solutions currently implemented in QUANTUM ESPRESSO introduce *several levels* of MPI parallelization:

1. *images* (points in configuration space)
2. *k-points*
2.5 *Kohn-Sham state* (under development)
3. *Plane Waves* (the first and the most important)
4. *Linear Algebra* (diagonalization etc)
5. *Kohn-Sham states in FFT* (so-called task groups)
(some localized special-purpose parallelizations are not counted)

and:
6. OpenMP.

Running on GPUs is also possible (but it is a quite different story)

Image parallelization

Images are currently one of the following objects:

1. a set of atomic positions in Nudged Elastic Band (NEB) calculations

2. atomic displacement patterns (irreps), used in linear-response (e.g. phonon) calculations.

Images are distributed across n_{image} groups of CPUs. Example for 64 processors divided into $n_{image} = 4$ groups of 16 processors each:

```
mpirun -np 64 neb.x -nimage 4 -inp input_file
```

(i.e. 4 NEB images are calculated in parallel on 16 processors each)

Image parallelization (2)

+ potentially linear CPU scalability, limited by number of images: n_{image} *must be a divisor of the number of NEB images/irreps*

+ very modest communication requirements: images typically communicate very little data at the end of the computation

o load balancing fair: unlikely that all images take the same CPU

– memory *is not distributed* at all!

Image parallelization is good when usable, especially if the communication hardware is slow

k-point parallelization

k-points are distributed (if more than one) among n_{pool} *pools* of CPUs. Example for 16 processors divided into $n_{pool} = 4$ pools of 4 processors each:

```
mpirun -np 16 pw.x -npool 4 -inp input_file
```

$+$ potentially linear scalability, limited by number of **k**-points: n_{pool} *must be a divisor of* N_k

$+$ modest communication requirements

o load balancing fair to good

– memory *is not distributed* in practice

Good if one has **k**-points, suitable for slow communication hardware

Band parallelization

Kohn-Sham states are split across the processors of the *band group*. Under development: a preliminary implementation exists for CP, where has similar effects as the task-group parrallelization (see later). In `pw.x`, works for exact-exchange and hybrid functionals only. n_{pool} *pools* of CPUs. Example for 64 processors divided into $n_{pool} = 2$ pools of 4 band groups of 8 processors each:

```
mpirun -np 64 pw.x -npool 2 -bgrp 4 -inp input_file
```

Plane-Wave parallelization

Wave-function coefficients are distributed among n_{PW} CPUs so that each CPU works on a subset of plane waves. The same is done on real-space grid points. This is the default parallelization scheme if other parallelization levels are not specified. In the example above:

```
mpirun -np 16 pw.x -npool 4 -inp input_file
```

plane-wave parallelization is performed on groups of 4 processors.

Plane-wave parallelization is historically the first serious and effective parallelization of PW-PP codes. The algorithm is rather nontrivial, requiring a parallel 3D FFT on distributed arrays with a specialized data structure in both R- and G-space.

It is still the parallelization “workhorse”.

Plane-Wave parallelization (2)

$+$ Memory is well distributed: in particular, all largest arrays are

$+$ load balancing is very good, provided n_{PW} *is a divisor of* N_3

$+$ excellent scalability, limited by the real-space 3D grid to $n_{PW} \leq N_3$

– heavy and frequent intra-CPU communications in the 3D FFT

Best choice overall, requires sufficiently fast communication hardware. Even so, it will saturate when $n_{PW} \sim N_3$, typically a few hundreds at most.

Linear-Algebra parallelization

Distribute and parallelize matrix diagonalization and matrix-matrix multiplications needed in iterative diagonalization (SCF) or orthonormalization (CP). Introduces a *linear-algebra group* of n_{diag} processors as a subset of the plane-wave group. $n_{diag} = m^2$, where m is an integer such that $m^2 \leq n_{PW}$. Should be set using the `-ndiag` or `-northo` command line option, e.g.:

```
mpirun -np 64 pw.x -ndiag 25 -inp input_file
```

Linear-Algebra parallelization (2)

+ RAM is efficiently distributed: removes a major bottleneck in large systems

+ Increases speedup in large systems

o Scaling tends to saturate: testing required to choose optimal value of n_{diag}

`./configure` selects `ScaLAPACK` (highly recommanded) if available; if not, home-made parallel algorithms are used. Useful in supercells containing many atoms, where RAM and CPU requirements of linear-algebra operations become a sizable share of the total.

Task-group parallelization

Each plane-wave group of processors is split into n_{task} task groups of n_{FFT} processors, with $n_{task} \times n_{FFT} = n_{PW}$; each task group takes care of the FFT over N_e/n_t states. Used to extend scalability of FFT parallelization. Not set by default. Example for 1024 processors divided into $n_{pool} = 4$ pools of $n_{PW} = 256$ processors, divided into $n_{task} = 8$ tasks of $n_{FFT} = 32$ processors each; subspace diagonalization performed on a subgroup of $n_{diag} = 144$ processors :

```
mpirun -np 1024 pw.x -npool 4 -ntg 8 -ndiag 144 ...
```

Allows to extend scaling when the PW parallelization saturates: definitely needed if you want to run on more than ~ 100 processors.

OpenMP parallelization

Explicit (with directives) parallelization (in particular of FFT: currently only for ESSL or FFTW) is activated at compile time with preprocessing flag (use `./configure --with-openmp ...`). Requires an OpenMP-aware compiler.
Implicit (with libraries) parallelization with OpenMP-aware libraries: currently only ESSL and MKL

+ Extends scaling

- Danger of MPI-OpenMP conflicts!

To be used on large multicore machines (e.g. IBM BlueGene), in which several MPI instances running on the same node do not give very good performances.

Summary of parallelization levels

group	distributed quantities	communications	performances
image	NEB images, phonon modes	very low	linear CPU scaling, fair to good load balancing; does not distribute RAM
pool	$\mathbf{k}$-points	low	almost linear CPU scaling, fair to good load balancing; does not distribute RAM
bands	KS orbitals	high	improves scaling
plane-wave	PW, **G**-vector coefficients, **R**-space FFT arrays	high	good CPU scaling, good load balancing, distributes most RAM
task	FFT on electron states	high	improves load balancing
linear-algebra	subspace hamiltonians and constraints matrices	very high	improves scaling, distributes more RAM
OpenMP	FFT, libraries	intra-node	extends scaling on multicore machines

State of implementation in the various packages

Not all parallelization levels are available for all packages in QUANTUM ESPRESSO. What is currently available:

- NEB: everything
- Self-consistent code, `pw.x`: k-points, plane-waves, tasks, linear-algebra, OpenMP, bands (for hybrid functionals)
- Car-Parrinello, `cp.x`: plane-waves, tasks, linear-algebra, OpenMP, bands (experimental)
- Linear-response code `ph.x`: images, k-points, plane-waves
- Other parallel codes: k-points, plane-waves

Most other codes can be run in parallel but they will execute on a single processor.

(Too) Frequently Asked Questions (that qualify you as a parallel newbie)

- How many processors can (or should) I use?
  It depends!

- It depends upon what??
  - Upon the kind of physical system you want to study: the larger the systems, the larger the number of processors you can or should use
  - Upon the factor driving you away from serial execution towards parallel execution: not enough RAM, too much CPU time needed, or both? If RAM is a problem, you need memory-distributing parallelizations, e.g. on plane waves
  - Upon your harware: plane-wave parallelization is ineffective on more than $4 \div 8$ processors with cheap communication hardware

- So what should I do???
  You should benchmark your job with different numbers of processors, pools, image, task, linear algebra groups, taking into account the content of previous slides, until you find a satisfactory configuration for parallel execution.

Compiling and running in parallel

- **Compilation**: if you have a working parallel environment (compiler / libraries), `configure` will detect it and will attempt a parallel compilation by default. Beware serial-parallel compiler conflicts, signaled by lines like

  WARNING: serial/parallel compiler mismatch detected

  in the output of `configure`. Check for the following in `make.sys`:

  `DFLAGS= ...-D__PARA -D__MPI ...`
  `MPIF90=mpif90` or any other parallel compiler

  For MPI-OpenMP parallelization, use
  `./configure --enable-openmp`
  verify the presence in file `make.sys` of `-D__OPENMP` in `DFLAGS`

If you plan to use linear-algebra parallelization, use

```
./configure --with-scalapack
```

verify the presence in file `make.sys` of `-D__SCALAPACK` in `DFLAGS`.
You may need to specify
`./configure --with-scalapack=intelmpi` or `...  =openmpi`.

If `configure` doesn't detect a parallel machine as such, your parallel environment is either disfunctional (nonexistent or incomplete or not working properly) or exotic (in nonstandard locations or for strange machines), or not properly set (like e.g. with the correct `module` command). Or maybe you simply found a case that `configure` doesn't know about.

Compiling and running in parallel (2)

- **Execution**: syntax is strongly machine-dependent!

  The total number of processor is either set by a *launcher* program: `mpirun` / `mpiexec` / `poe` / ... or by the batch queueing system (NQS/PBS/LoadLeveler/...)

  Options `-nimage`, `-npool`, `-ndiag`, `-ntg` are interpreted by QUANTUM ESPRESSO executables and should follow them.

  You may need (in future versions, you *will have*) to supply input data using QUANTUM ESPRESSO option "`-inp` *filename*" instead of "$<$ *filename*" ; you may also need to supply optional arguments via "`mpirun -args '`*optional-arguments*`'`"

Compiling and running in parallel (3)

- **Tricks and Tips**:
  Be careful not to make heavy I/O via NFS (Network File System). Write to either a parallel file system (present on expensive machines) or to local disks. In the latter case, check that all processors can access `pseudo_dir` and `outdir`. SCF: use option `wf_collect` to collect final data files in a single place; consider using option `disk_io` to reduce I/O to the strict minimum.

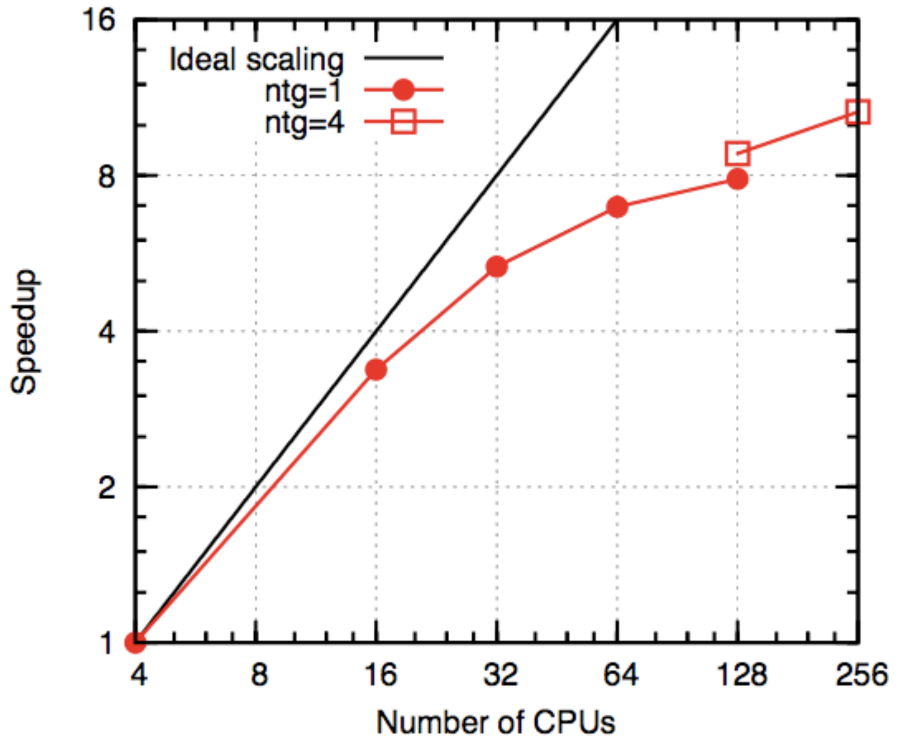
  Mixed MPI-OpenMP parallelization should be used ONLY on machines (e.g. BlueGene) that allow you to control how many MPI processes run on a given node, and how many OpenMP threads.

Scalability for “small” systems

Scalability *strongly depends upon the kind of system under examination*! Max number of processors for satisfactory scalability limited by system “size”, i.e. number of atoms, number of electrons, dimensions of the simulation cell. *Easy*’ parallelization, e.g. of images, not considered here.

Typical speedup vs number of processors:

128 Water molecules (1024 electrons) in a cubic box 13.35 A side, Γ point.
`pw.x` *code on a SP6 machine, MPI only.*
ntg=1 parallelization on plane waves only
ntg=4 also on electron states

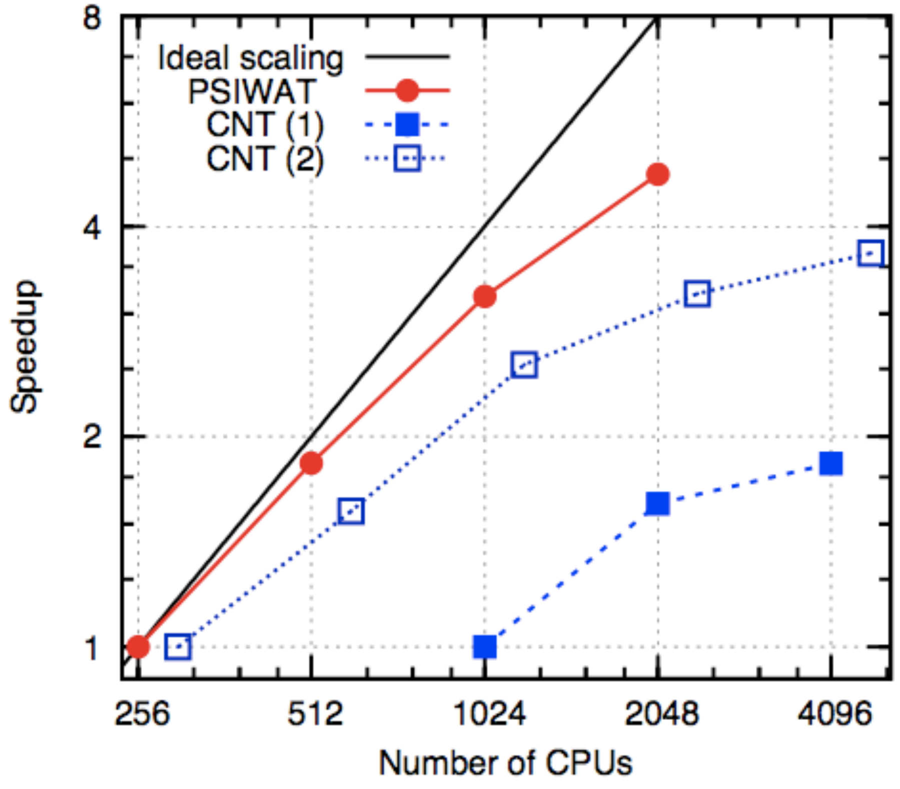


Scalability for "medium-size" systems

PSIWAT: Thiol-covered gold surface and water, 4 $\mathbf{k}-$*points,* $10.59 \times 20.53 \times 32.66\ A^3$ *cell, 587 atoms, 2552 electrons.* `pw.x` *code on CRAY XT4, parallelized on plane waves, electron states,* $\mathbf{k}-$*points. MPI only.*

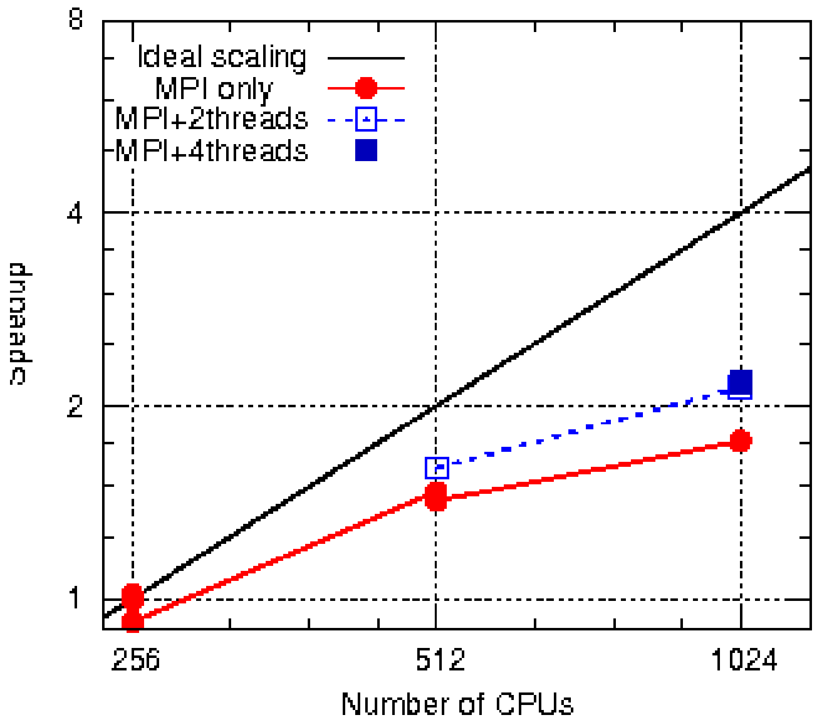
CNT(1): nanotube functionalized with porphyrins, Γ *point, 1532 atoms, 5232 electrons.* `pw.x` *code on CRAY XT4, parallelized on plane waves and electron states, MPI only.*

CNT(2): same system as for CNT(1). `CP` *code on a Cray XT3, MPI only.*



3D parallel distributed FFT is the main bottleneck. Additional parallelization levels (on electron states, on $\mathbf{k}-$points if available) allow to extend scalability.

Mixed MPI-OpenMP scalability



Fragment of an Aβ-peptide in water containing 838 atoms and 2312 electrons in a 22.1×22.9×19.9 A^3 cell, Γ-point. `CP` code on BlueGene/P, 4 processes per computing node.

Two models of graphene on Ir surface on a BlueGene/P using 4 processes per computing node. Execution times in s, initialization + 1 self-consistency step.

N cores	T cpu (wall) 443 atoms	T cpu (wall) 686 atoms
16384	740(772)	2861 (2915)
32768	441(515)	1962 (2014)
65536	327(483)	751 (1012)

(the large difference between CPU and wall time is likely due to I/O)