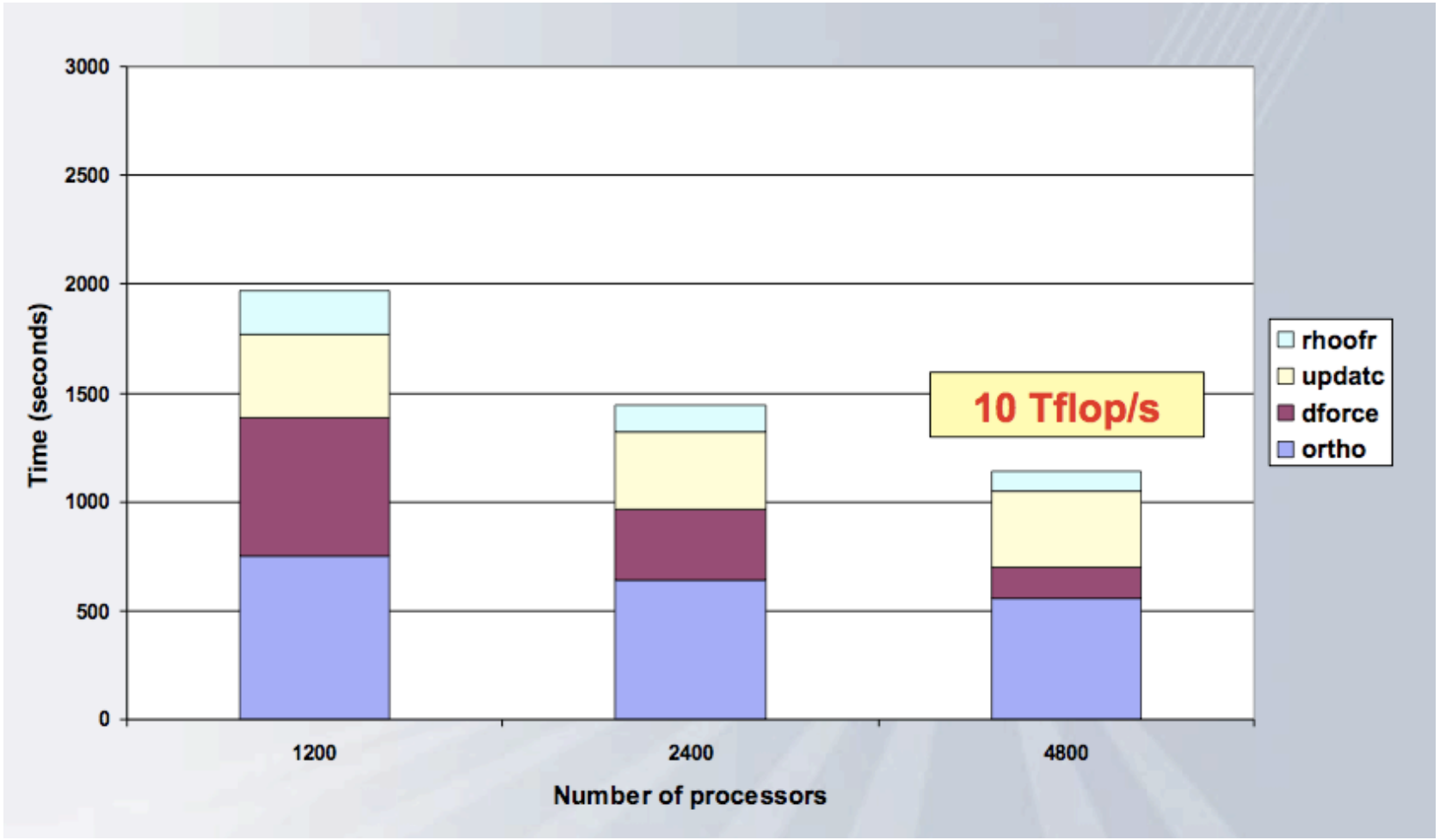
Quantum-ESPRESSO

Notes on parallel computing



Parallel programming paradigms

Most modern electronic-structure code use one of the following approaches, or a combination of them:

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

- **OpenMP parallelization**
  A single process spawns sub-processes (or "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.

Currently quantum-ESPRESSO uses MPI parallelization; it can achieve some degree of OpenMP parallelization via mathematical libraries, but beware MPI-OpenMP conflicts!

Comparison of parallel programming paradigms

MPI parallelization:

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

+ Very efficient (as long as MPI libraries are)

+ Can scale up to a large number of processors

– Requires significant code reorganization and rewriting

OpenMP parallelization:

+ Relatively easy to implement: requires modest code reorganization and rewriting

– Not very efficient: doesn't scale on more than a few processors, interesting only for multi-core CPU's.

Relevant variables in PW-PP electronic-structure codes

N_w: number of plane waves (used in wavefunction expansion)
N_g: number of **G**-vectors (used in potential/charge density expansion)
N_1, N_2, N_3: dimensions of the FFT grid[1]
N_a: number of atoms in the unit cell or supercell
N_e: number of electron (Kohn-Sham) states (bands) used in the calculation
N_p: number of projectors in non-local pseudopotentials (summed over the cell)
N_k: number of **k**-points in the irreducible Brillouin Zone

Note that

$$N_w \propto V \quad \text{(size of the unit cell)}, \quad N_w \propto E_c^{3/2} \quad \text{(kinetic energy cutoff)}$$

and

$$N_g \sim N_1 \times N_2 \times N_3, \quad N_g \sim \alpha N_w$$

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

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

[1] Note that there can be two distinct FFT grids with US PP

Time-consuming steps in PW-PP electronic-structure codes

- Calculation of density, $n(\mathbf{r}) = \sum |\psi(\mathbf{r})|^2$:
  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 in PW-PP electronic-structure codes

- **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. If $n_k > 1$, arrays can be stored to disk and used one at the time, thus reducing memory usage, at the price of increased disk I/O

- **Charge density and potential**:
  several vectors with dimension N_g (in **G**-space) or $N_r = N_1 \times N_2 \times N_3$ (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 slower or much slower than computation!

- **Distribute all memory that grows with the size of the cell**:
  if you don't, you will run out of memory for large cells

- **Distribute all computation that grows with the size of the cell**:
  if you don't, sooner or later non-parallelized calculations will take most of the time (a practical demonstration of Amdhal's law)

The solution currently implemented in Quantum-ESPRESSO introduces *several levels of parallelization*.

Parallelization level 1

- **Image** parallelization:
  Currently implemented only for NEB, but there are other cases where it could be useful. Images, i.e. points in the coordinate space, 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 pw.x -nimage 4 -inp input_file
  ```

  \+ potentially linear CPU scalability, limited by number of images:
  n_{image} *must be a divisor of the number of NEB images*
  \+ very modest communication requirements

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

  – memory usage *does not scale* at all!

Good when usable, especially if the communication hardware is not so fast

Parallelization level 2

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

  ```
  mpirun -np 64 pw.x -nimage 4 -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 usage *does not scale* in practice

Good if one has **k**-points, especially if the communication hardware is not so fast

Parallelization level 3

- **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 64 pw.x -nimage 4 -npool 4 -inp input_file
  ```

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

  \+ high scalability of memory usage: the largest arrays are all distributed
  \+ good load balancing among different CPUs *if* 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, mainly in the 3D FFT

Best choice overall, if one has sufficiently fast communication hardware

Parallelization levels 4 and 5

- **linear-algebra** parallelization:
  distribute and parallelize matrix diagonalization and matrix-matrix multiplications needed in iterative diagonalization (PWscf) 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 $m^2 \leq n_{PW}$. Set by default to the largest possible value; if not appropriate, can be changed to a smaller value using the `-ndiag` or `-northo` command line option, e.g.:

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

- **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. Example for 4096 processors divided into $n_{image} = 8$ images of $n_{pool} = 2$ 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 4096 pw.x -nimage 8 -npool 2 -ntg 8 -ndiag 144 ...
  ```

Summary of parallelization levels in Quantum-ESPRESSO

group	distributed quantities	communications	performances
image	NEB images	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
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

(Too) Frequently Asked Question
(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 plane-wave parallelization
  - Upon the kind of machine you have: plane-wave parallelization is ineffective on more than $4 \div 8$ processors connected 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
  ```

  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) .

Specific to this tutorial: use `./configure F90=gfortran` to prevent conflicts between the Intel and `gfortran` compilers

Compiling and running in parallel (2)

- **Execution**: beware the syntax! the total number of processor is an option of `mpirun`, `mpiexec` or whatever applies (or of the batch queue); `-nimage`, `-npool`, `-ndiag`, `-ntg` are options of Q-E executables and should follow them.

  Be careful not to make heavy I/O via NFS. Write to either a parallel file system (found only on expensive machines) or to local disks. In the latter case, check that all processors can access `pseudo_dir` and `outdir`. `PWscf`: use option `wf_collect` to have the final data files in a single place and not scattered across different machines; consider using option `disk_io` to reduce I/O to the strict minimum.

  Beware autoparallelizing libraries (e.g. Intel MKL or IBM ESSL): multithreading must be explicitly disabled by setting `OMP_NUM_THREADS` to 1, or else MPI and OpenMP will spend their time fighting for the same CPU. In general: check that there aren't other processes interfering with yours.

Specific to this tutorial: some machines are connected as a cluster, others may do MPI only on the two cores of a single machine.

Your first parallel calculation with quantum-ESPRESSO

- make a copy of Q-E (e.g. in `espresso-para/`), compile `pw.x` for parallel execution (you will need MPI libraries installed and working)
- get the (rather bogus and unphysical) example in `/root/Q-E/PARA/`
- run it first on one processor, then on two processors with one or two pools, and if you can, on four processors in various modes. What do you observe?

PS: Don't expect a major speedup!