CEA / DAM - Île de France / Département Sciences de la Simulation et de l'information / Service Informatique Scientifique et Réseaux

PARALLELIZATION

IN ABINIT

PLAN

- ✓ My activities
- ✓ History of parallelization in Abinit
- ✓ First method : MPI
- ✓ Second method : OpenMP
- ✓ Implementation in Abinit of MPI
- ✓ Implementation in Abinit of OpenMP
- ✓ MPI versus OpenMP
- ✓ Evolutions
- ✓ Conclusion

My activities

Working in France near Paris : CEA

2 main activities :

✪ DBA : DataBase Administrator (Oracle Software).

✪ Parallelization of codes : use of a Compaq computer (EV68) of about 2400 processors (largest computer in Europe, 4th in the world).

History of parallelization in Abinit

➢ Beginning of the collaboration in November 1999. Some parallelization was already in place (kpoints in ground state and fundamental cases) ➟ introduction of a new parallelization level on bands with MPI in ground state case,

➢ Optimization of this parallelization on read/write on files (March 2000 - May 2000),

➢ Introduction of OpenMP directives (May 2000 - January 2001),

➢ New parallelization : on nsppol in ground state case (February 2001 - March 2001),

History of parallelization in Abinit

➢ New structure of Abinit : introduction of modules, of types : upgrade of parallelization with mpi_defs module and mpi_type structure (November 2001),

➢ New parallelization level : on groups of bands in fundamental state case (december 2001),

➢ Implementation of a new feature : parareel : parallelization of calls of gstate (april - july 2002).

First method : MPI

➢ MPI is for Message Passing Interface,

➢ Standard, portable method for all scientific computers. The API is standard but the implementation is constructor-dependent,

➢ In this method, all the processors execute the same code, and send/receive messages to/from each other in order to communicate results.

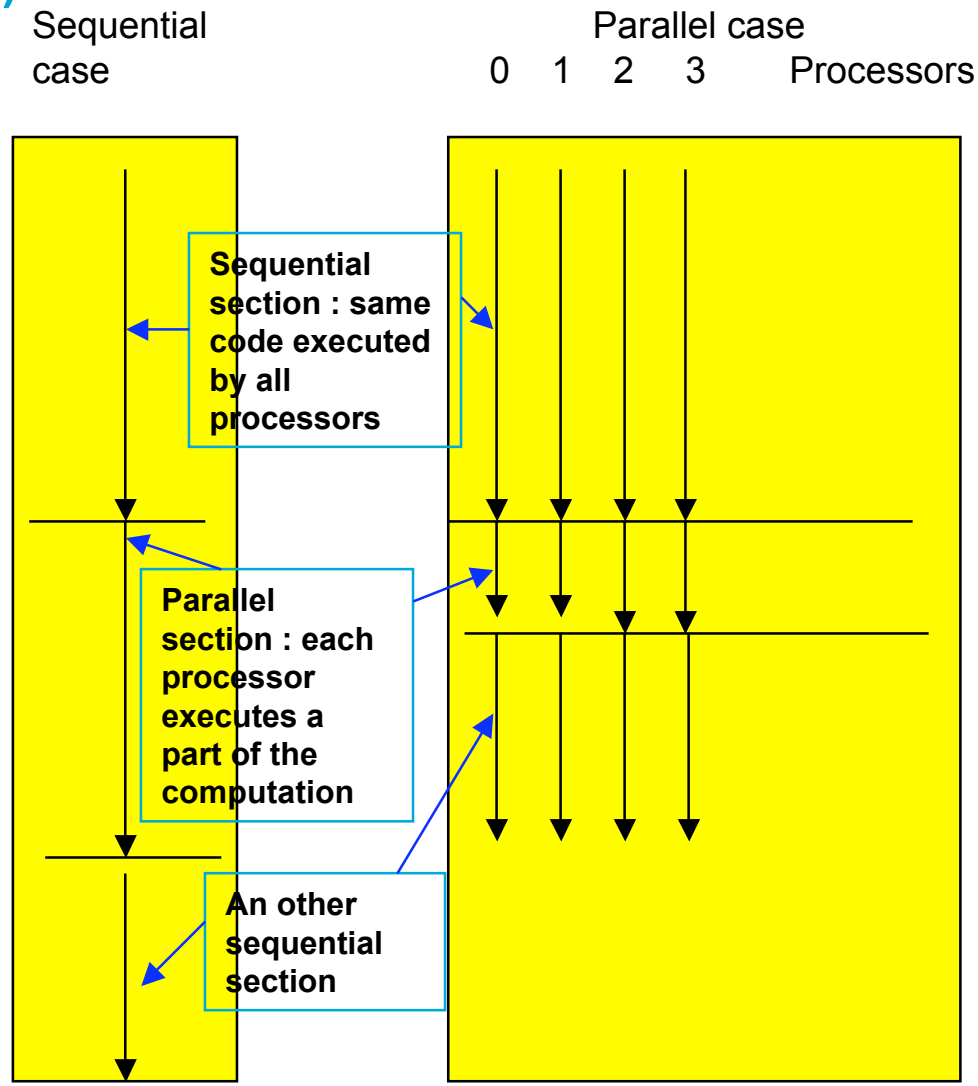
PARALLELIZATION IN ABINIT
First method : MPI (example 1)

Example 1 :

All the codes have some sequential sections and parallel sections.

A sequential section is a group of instructions that all processors have to do (initializations ...).

A parallel section is a work that can be divided between a small or large number of processors : each processor makes a part of the work. At the end, they have usually to synchronize their results in order to continue the code.



Some explanations :

The gain is only in the parallel section ...

The work to do has to be « independent » : if the parallel section is a loop, each loop iteration can be executed by itself. It does not need the result of another loop iteration.

The parallel section is generally a loop. Each processor has to do some iterations of the loop. Caution : each processor works with its own variables. At the end of the loop, the developer may have to add a « synchronization barrier» in order to continue the computation correctly. An example of this :

PARALLELIZATION IN ABINIT
First method : MPI (example 2)

```
do i=1, 100
        A=A+tab[i]
enddo
```

Sequential Version :
The processor makes 100 additions

```
do i=1, 100
        if  (it is my work) then
        A=A+tab[i]
        endif
enddo
call MPI_ALLREDUCE
(A,A_tot,1,MPI_INTEGER,MPI_SUM,
MPI_COMM_WORLD,IERR)
A=A_tot
```

Parallel Version :
Each processor makes 100/nproc additions

Implicit synchronization :
adds the all sums and sends the result to all processors

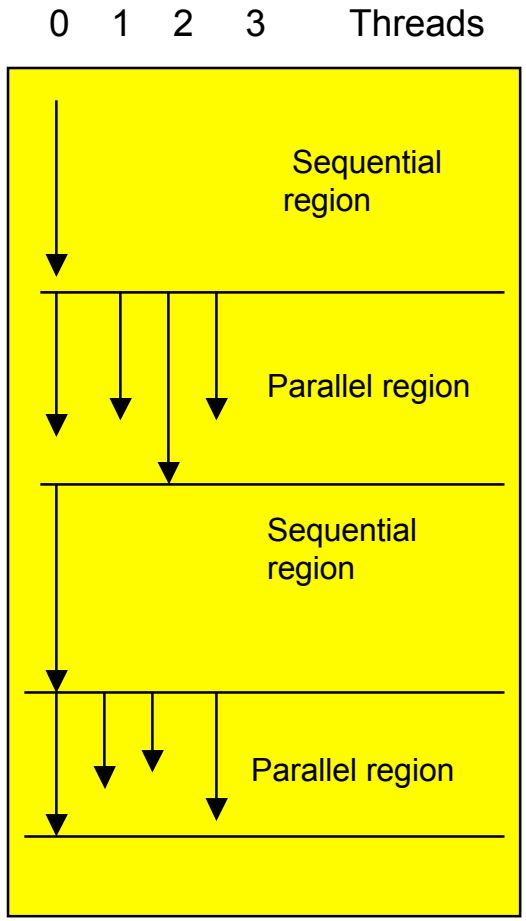
Second method : OpenMP

➢ OpenMP is a parallelization method by insertion of directives,

➢ Standard, portable method for all shared memory computers. The directives are standards but the implementation is constructor-dependent,

➢ In this method, only one processor executes the original code. In the parallel regions, threads are created : they share the variables (via the shared memory) and the work to do.

PARALLELIZATION IN ABINIT
Second method : OpenMP



✓ An OpenMP program is a sequence of parallel and sequential regions

✓ A sequential region is always executed by the master thead (thread 0)

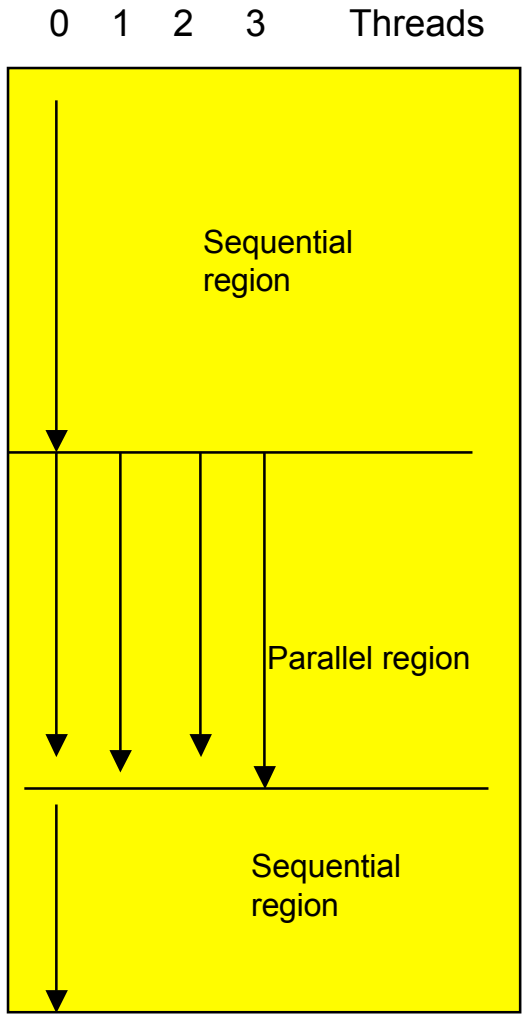
✓ A parallel region may be executed by many threads at the same time

✓ The number of threads doesn't vary in a parallel region

PARALLELIZATION IN ABINIT
Second method : OpenMP

Example :

```
        …
        REAL A(N)
        …
        A=0
!$OMP PARALLEL DO PRIVATE(I) SHARED(A,N)
        DO I=1, N
                A(I)=A(I)+1
        END DO
!$OMP END PARALLEL DO
        …
```



Implementation of MPI in Abinit

- Use of module defs_mpi
- Use of routine distrb2 / array proc_distrb
- Example of modifications into Abinit Code
- Performances
- Conclusion

Use of module defs_mpi

➢ This module contains all the variables useful for MPI parallelization. These variables are intialized in the distrb2 routine.

➢ Examples of variables :

✓ nproc : number of processors who participate in the run,

✓ me : rank of the current processor in the list of processors (0 to nproc-1),

✓ paralbd : 0 parallelization on kpoints, 1 on bands, > 1 on blocks on bands,

✓ some integer variables and arrays to manage the communication between the processors groups.

Use of routine distrb2 / array proc_distrb

➢ This routine is called at the beginning of the code, in order to initialize the array proc_distrb. This array contains the numbers of the processor which have to treat each iteration :

proc_distrb(ikpt,iband,isppol)= iproc : all the iterations about the band iband of the kpoint ikpt of the nsppol isppol have to be treated by the processor iproc.

➢ In distrb2, the array proc_distrb is initialized with a different method based on the paralbd variable. The user can also create a file with his own repartition (kpt_distrb file). Some messages could appear if the number of processors is not adequate with the problem (nproc too large, not a multiple of a variable ...).

Use of routine distrb2 / array proc_distrb

An example of the work repartition into a loop : 10 processors, 6 kpoints and 5 bands :

kpoint \ band	band 1	band 2	band 3	band 4	band 5
1	0	1	2	3	4
2	5	6	7	8	9
3	0	1	2	**3**	4
4	5	6	7	8	9
5	0	1	2	3	4
6	5	6	7	8	9

The band 4 of the kpoint 3 has to be treated by the processor 3.

Examples of modifications into Abinit Code (vtowfk3.f)

```
 use defs_mpi
...
#         if defined MPI
          include 'mpif.h'
#         endif
...
!This type is defined in defs_mpi
 type(MPI_type) :: mpi_enreg
...
#         if defined MPI
          !Variables introduced for MPI version
          integer :: ierr,me
#         endif
...
!Loop over bands
 do iband=1,nband_k
#         if defined MPI
          if(mpi_enreg%proc_distrb(ikpt, iband,isppol) /= mpi_enreg%me) then
                              cycle
          endif
#         endif
...
enddo
```

Performances

Test_paral #6 on a Compaq ES40 computer :

	1 proc	2 procs	4 procs	10 procs
total wall	90.8	64.0	44.4	33.2
max theoric total wall		55.58	38.07	27.56
(seq-time + (parallel-time on 1 proc / nproc))				
vtorho3	70.04	41.42	22.33	11.49
vtowfk3	64.91	34.87	17.88	6.97
speedup vtowfk3		**1.86**	**3.63**	**9.31**
total speedup		1.42	2.05	2.73
max theoric total speedup		1.63	2.39	3.29

Conclusion

✓Implementation of MPI parallelization needs to know very well the variables and the processing of the code

✓ Sometimes difficult to resolve bugs in parallel

✓Optimization operations were introduced to improve speedup

✓The efficency of the parallelization is rather easy to demonstrate (a lot of kpoints or bands in a test case).

Implementation of OpenMP in Abinit

- Examples of modifications into Abinit Code
- Performances
- Conclusion

Example of modifications into Abinit Code (in vtowfk3.f)

```
...
!$OMP PARALLEL DO PRIVATE(ipw) &
!$OMP&SHARED(cwave0,cwavef_sp,npw_k)
      do ipw=1,npw_k
       cwavef_sp(1,ipw)=cwave0(1,ipw+npw_k)
       cwavef_sp(2,ipw)=cwave0(2,ipw+npw_k)
      enddo
!$OMP END PARALLEL DO
...
!$OMP PARALLEL DO PRIVATE(i1,i2,i3) &
!$OMP&SHARED(n1,n2,n3,rhoaug1,weight,wfraug,wfraug1)
      do i3=1,n3
       do i2=1,n2
        do i1=1,n1
         rhoaug1(i1,i2,i3)=rhoaug1(i1,i2,i3)+&
&            weight*( wfraug(1,i1,i2,i3)*wfraug1(1,i1,i2,i3) &
&                  +wfraug(2,i1,i2,i3)*wfraug1(2,i1,i2,i3)  )
        enddo
       enddo
      enddo
!$OMP END PARALLEL DO
...
```

Performances into Abinit Code

Test_paral #3 (with ecut=100 -> npw > 42000) on a Compaq ES40 computer : (Time in seconds)

	1 thread	2 threads	4 threads
Total Wall	729.6	471.6	315
fourwf(pot)	355.18	236.32	133.25
projbd	92.07	45.38	25.18
nonlop(apply)	75.28	44.87	33.8
fourwf(den)	26.31	17.43	9.89
Total Speedup		**1.55**	**2.32**
fourwf(pot) Speedup		1.5	2.67
projbd Speedup		**2**	**3.66**
nonlop(apply) Speedup		1.68	2.23
fourwf(den) Speedup		1.51	2.66

Conclusion

✓ Easy to insert directives in the code

✓ Difficult to debug, but very few bugs to find

✓ Very difficult to optimize and to demonstrate efficency (some parallelization of loops had to be suppressed) : a lot of possibilities with environment variables and options in directives

✓ Could have some memory and cache problems (faults)

✓ Some optimizations could be machine-dependent

Performances MPI versus OpenMP

Example Test_paral #3 with ecut=100 (nkpt=4, npw > 42000) on a Compaq ES40 computer (fundamental case) :

MPI Processors

Threads OpenMP	1	2	4
1	729.6	390.6	250.6
Speedup		1.87	2.91
2	471.6	262.7	151.7
Speedup	1.55	2.78	4.84
4	315	179.2	**109**
Speedup	2.32	4.07	**6.69**

Evolutions

✓ Parallelization on bands in the fundamental case (just on groups of bands actually)

✓ Modifications of algorithms ??

✓ Use of MPI-IO for the read/write of files ??

Conclusion

✓ The use of the different method of parallelization for a case depend on the values of nkpt/nband and npw on one hand, and on the number of processors you have on an other hand.

✓ MPI is easier to understand :maximum number of processors to use depends on value of nkpt (and nband in fundamental case) ; the parallelization is predictable

✓ OpenMP could be useful when you have much more processors than nkpt.

For more information

- www.openmp.org
- www.mpi-forum.org
- www.idris.fr/data/cours/parallel/openmp
- www.idris.fr/data/cours/parallel/mpi