MPI: a short introduction

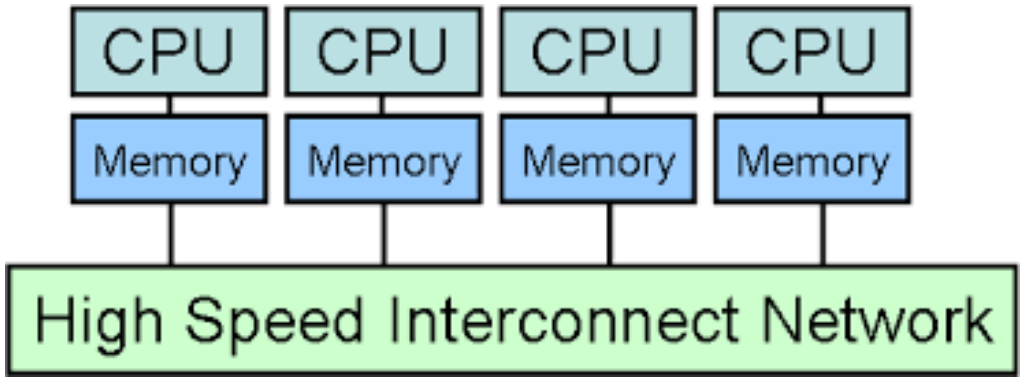
Stefano Gandolfi

Los Alamos National Laboratory (LANL)

TALENT School on Nuclear Quantum Monte Carlo Methods
North Carolina State University (NCSU), July 11-29 2016.

MPI: Introduction

Given the nature of Monte Carlo algorithms, it is often very easy to use parallel machines or supercomputers. What are they? They are thousands (or more) cpus interconnected with fast network:



Ideally, if we have M (independent) configurations to sample, taking a time t using one computer, using N cpus we might reduce the time to t/N. This might be easy by spreading the M configurations over the N cpus!

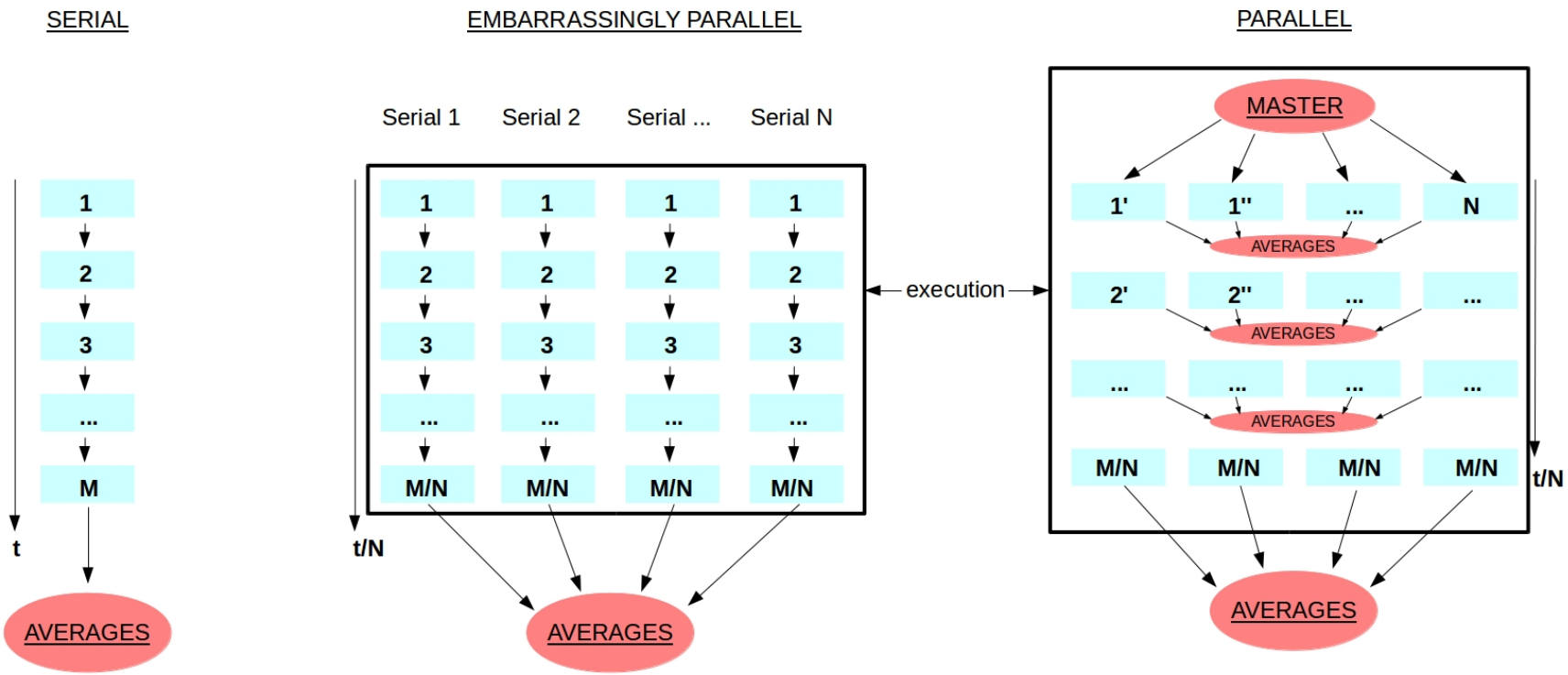
The most common libraries used to "parallelize" a code are the MPI = Message Passing Interface

MPI: Introduction

Some definition:

- Serial code/run: running on one single cpu
- Embarrassingly parallel: basically no communication needed between cpus
- Parallel code/run: can run using 1,2,...,N cpus that need to communicate

MPI: Introduction



MPI: setup

The usual way to run a code is something like:

```
-bash$ ./exec < input > output
```

With MPI (OpenMPI), the way is instead:

```
-bash$ mpirun -np N ./exec < input > output
```

where N is the number of MPI-threads that you use.

Then the job will have N cpus available.

MPI: setup

How to use the N available MPI-threads?

```
...
call mpi_init(ierror)
call mpi_comm_rank(mpi_comm_world,irank,ierror)
call mpi_comm_size(mpi_comm_world,iproc,ierror)
...
```

The above lines are the first step to initialize an MPI code.

MPI: setup

```
call mpi_init(ierror)
```

Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

```
call mpi_comm_rank(mpi_comm_world,irank,ierror)
```

Returns the rank of the calling MPI process within the specified communicator. **mpi_comm_world** is the ID of the predefined communicator that includes all MPI processes (master). **irank** is an integer referred to as a task ID (one for each task).

```
call mpi_comm_size(mpi_comm_world,iproc,ierror)
```

Returns the total number **iproc** of MPI processes in the specified communicator.

MPI: main subroutines

```
mpi_send(data,size,datatype,dest,tag,comm,ierr)
```

Send the **data** (can be an array) of size **size** of type **datatype** to the rank **dest**.

```
mpi_recv(data,size,datatype,source,tag,comm,status,ierr)
```

Receive the **data** (can be an array) of size **size** of type **datatype** from the rank **source**.

```
mpi_bcast(data,size,datatype,root,comm,ierr)
```

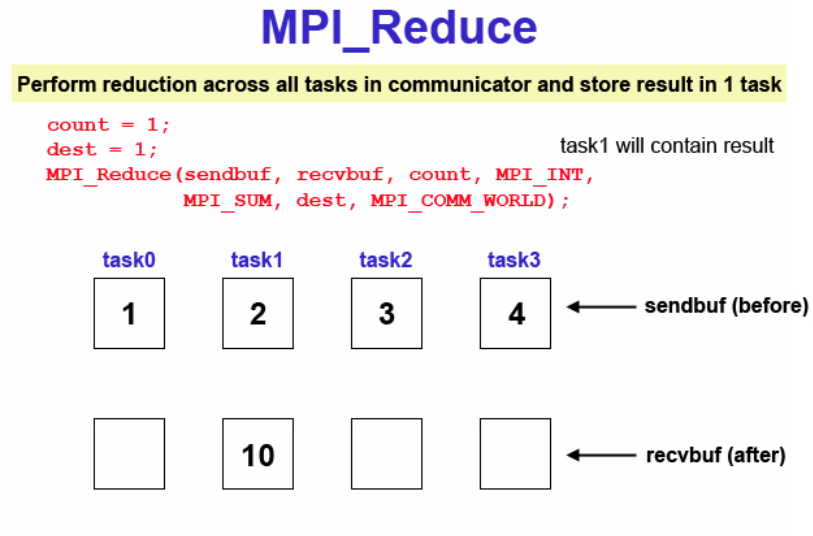
Task **root** send **data** (can be an array) of size **size** of type **datatype** to all the other ranks.

Most used **datatypes**: **mpi_integer**, **mpi_double_precision**, **mpi_double_complex**.

MPI: main subroutines

```
mpi_reduce(senddata,recvdata,size,datatype,op, &
           root,comm,ierr)
```

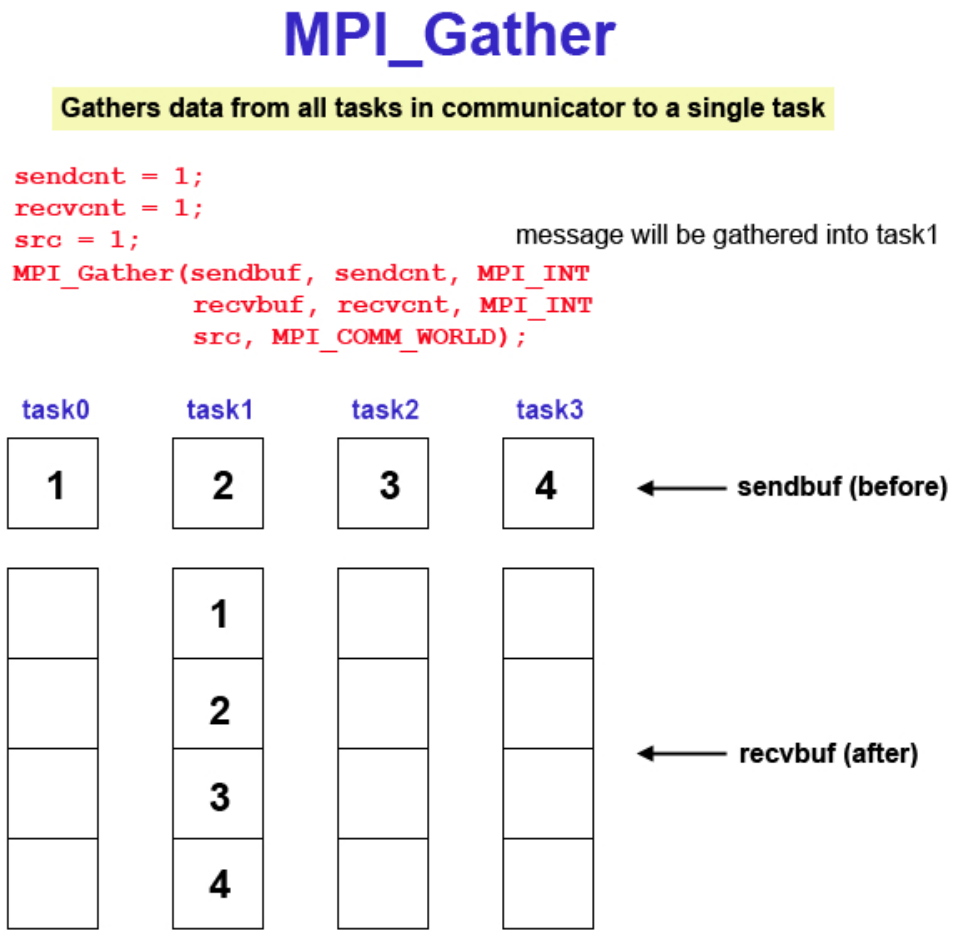
Collective computation operation. Applies a reduction operation on all tasks in the group and places the result in one task **root**. For example: operation **mpi_sum**, each rank send **senddata** of size **size** to the rank **root**. The latter will store in **recvdata** the **sum** of the received data.



MPI: main subroutines

```
mpi_gather (senddata,sendsize,sendtype,recvdata, &
     recvsize,recvtype,root,comm,ierr)
```

Gathers distinct messages from each task in the group to a single destination task.



MPI: first example

```
    call mpi_init(ierror)
    call mpi_comm_rank(mpi_comm_world,irank,ierror)
    call mpi_comm_size(mpi_comm_world,iproc,ierror)
    if (irank.eq.0) write (6,'(''I am the master node. The total number of CPUS is  '',i10)
    if (irank.eq.0) ntab=10000000  ! only the master (id=0) does this
    call mpi_bcast(ntab,1,mpi_integer,0,mpi_comm_world,ierror) ! send to the others
...
    if (irank.eq.0) then
       do i=1,ntab
          q(:)=0.001*i
          f1(i)=blackbox(q) ! buildup some data
       enddo
    endif
! split the work to do, each cpu will operate on some part of the big array
    ntabcpu=ntab/iproc
    n0=ntabcpu*irank+1     ! this is the first point
    n1=ntabcpu*(irank+1)   ! this is the last point
    call cpu_time(time0)
    do i=n0,n1
       f2(i)=blackbox(q)
    enddo
    call cpu_time(time1)
    t2=time1-time0
    write(6,'(''myID, time: '',i10,f15.5)') irank,t2
! now the master collects all the data to print
    call mpi_reduce(f2,fsum,size(f2),mpi_double_precision,mpi_sum,0,mpi_comm_world,ierror)
...
```

MPI: first example

```
$ mpirun -np 1 ./exec
I am the master node. The total number of CPUS is    1
myID, time:           0        2.03368


$ mpirun -np 10 ./exec
I am the master node. The total number of CPUS is   10
myID, time:           1        0.20690
myID, time:           7        0.20550
myID, time:           6        0.20637
myID, time:           8        0.20662
myID, time:           9        0.20579
myID, time:           3        0.20425
myID, time:           4        0.20552
myID, time:           5        0.20477
myID, time:           0        0.20469
myID, time:           2        0.20729
```

MPI: load balancing

The parallelization of DMC is not trivial as VMC. Can you guess why it is different from VMC?

MPI: load balancing

The parallelization of DMC is not trivial as VMC. Can you guess why it is different from VMC?

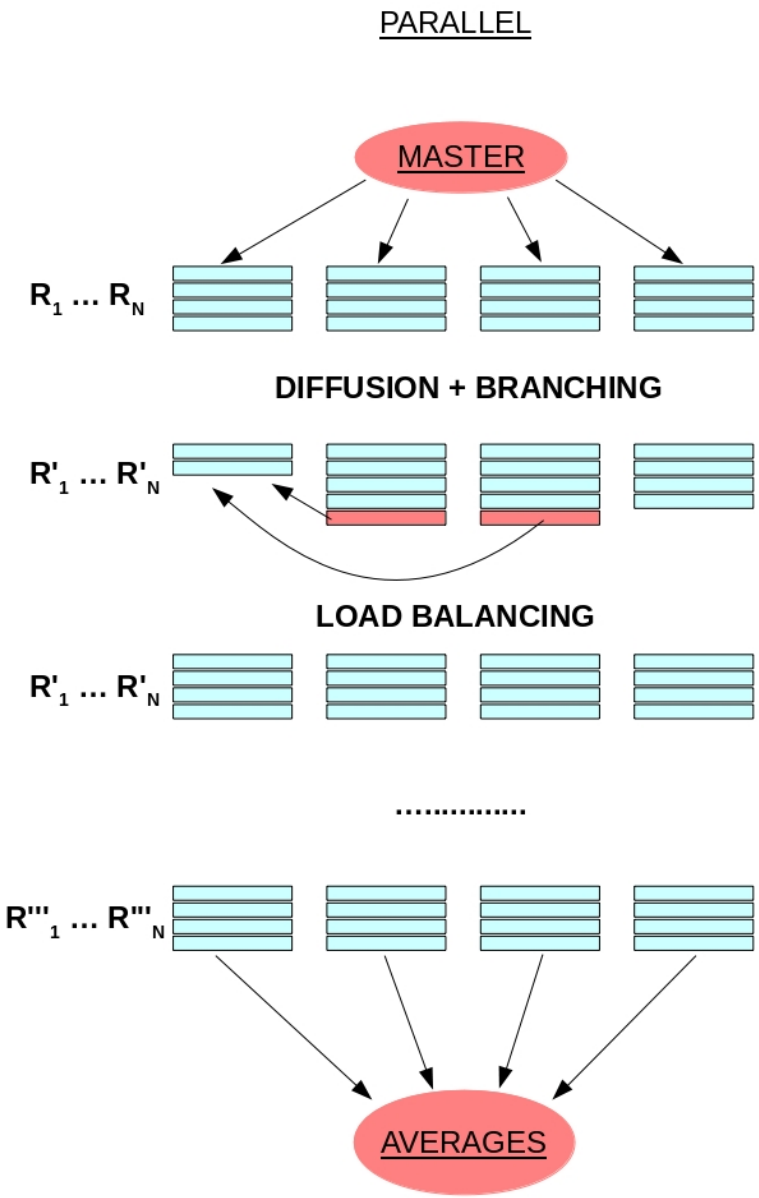
Branching changes the number of configurations dynamically!

Load balancing: redistribute the configurations among cpus in order to keep the same working load (as much as possible).

Bottleneck: when moving data becomes less convenient than having some cpu doing more work.

MPI and DMC

Load balancing:



MPI and random numbers

One thing to be extremely cautions when writing a parallel code is about random numbers.

Suppose that we send two identical configurations to two cpus using the same random seed, than the two configurations will have the exact same evolution through the execution!
We can still send the same configuration to two cpus, just use different random seeds.

The argument above also applies while doing branching. In this case if we have to duplicate a configuration, just make sure that the two branched configurations have different random seeds.

One possible implementation

My way of implementing things is to define a *structure* defining a walker:

```
module stack
   implicit none
   type :: walker
      real :: x,psi,dpsi,d2psi
      real :: vext,weight, ...
      integer :: irn
   end type
   type(walker), allocatable :: s(:,:)
contains
   subroutine copywalker(wl,wr)
   type(walker), intent(inout) :: wl
   type(walker), intent(in) :: wr
   wl%x=wr%x
   wl%psi=wr%psi
   wl%dpsi=wr%dpsi
   wl%d2psi=wr%d2psi
   wl%vext=wr%vext
   wl%weight=wr%weight
   wl%irn=wr%irn
   ...
   end subroutine copywalker
```

```
   subroutine push(i,w)
   integer :: i
   type(walker) :: w
   ist(i)=ist(i)+1
   s(ist(i),i)=w
   end subroutine push

   subroutine pop(i,w,empty)
   integer :: i
   type(walker) :: w
   logical :: empty
   empty=ist(i).eq.0
   if (.not.empty) then
      w=s(ist(i),i)
      ist(i)=ist(i)-1
   endif
   end subroutine pop
end module stack
```

Each walker has its own random seed (irn)

One possible implementation

Walkers are moved between cpus using something like:

```
subroutine movewalkers(ifrom,ito)
...
if (ifrom.eq.ito) return
if (irank.eq.ito) then ! who receives
   do i=1,nwalk
      call mpi_recv(w%x,1,mpi_double_precision,ifrom,id, &
         mpi_comm_world,istatus,ierror)
  ...
      call mpi_recv(w%irn,1,mpi_integer8,ifrom,id, &
         mpi_comm_world,istatus,ierror)
      call push(istack,w)
   enddo
else if (irank.eq.ifrom) then ! who sends
   do i=1,nwalk
      call pop(istack,w,empty)
      call mpi_send(w%x,1,mpi_double_precision,ito,id, &
         mpi_comm_world,ierror)
  ...
      call mpi_send(w%irn,1,mpi_integer8,ito,id, &
         mpi_comm_world,ierror)
   enddo
endif
```

Introduction to MPI

MPI very useful when parallel machines are available!

Introduction to MPI

MPI very useful when parallel machines are available!

Please, don't panic (yet...), some example in the afternoon :-)

Introduction to MPI

MPI very useful when parallel machines are available!

Please, don't panic (yet...), some example in the afternoon :-)

... for now :-)