HPC Tools 2024, Exercise 8 solutions

Bruno Oliveira Cattelan, Laurent Chôné

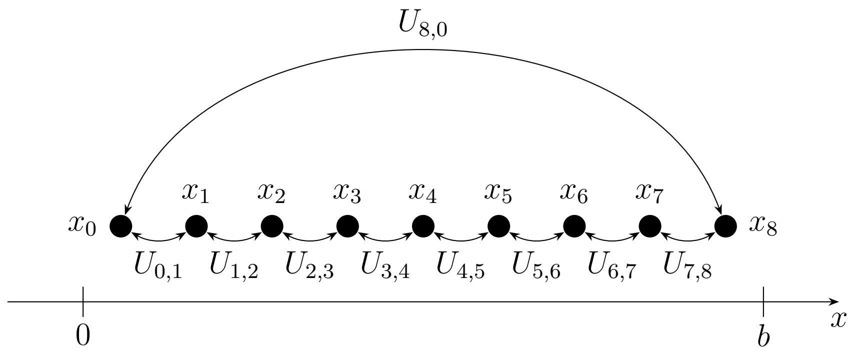
Problem 1

In this problem, we need to parallelise an existing dynamical simulation code (provided). The physical system consists of a periodic chain of particles arranged on a 1-dimensional line, which interact with their nearest neighbours.

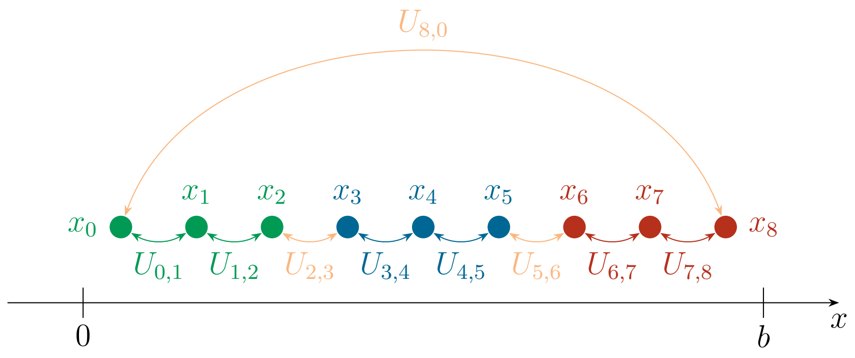
For more clarity, let us define the interaction potentials between particles. The potential of particle i is:

$$\begin{aligned} U_i =& \frac{k_1}{2}\left(x_i - x_{i-1} - d\right)^2 + \frac{k_2}{2}\left(x_i - x_{i-1} - d\right)^3 + \frac{k_1}{2}\left(x_{i+1} - x_i - d\right)^2 + \frac{k_2}{2}\left(x_{i+1} - x_i - d\right)^3 \\ =& U_{i,i-1} + U_{i+1,i} \end{aligned}$$

Hence, at a fixed instant in time, the situation is as below (illustrated with 9 particles), where the arrows indicate the interaction potentials between neighbouring particles.



It is assumed in the problem statement that the graph topology of the system does not change during the simulation (meaning particle never swap positions), therefore the system may be decomposed by dividing the particles between ranks (assuming 3 processes):



Now particles of a same color are owned by the same process, and interaction potentials between those particles can be calculated independently. Interaction potentials between particles belonging to different processes (orange arrows) require that those processes exchange the position information of the particles involved.

Initialisation is distributed between processes, therefore in C++:

```
//*** C++ ***

double box=nat;
srand(time(NULL));
for (int i=0;i<nat;i++)
{
    x[i]=i;
    rn=(double)rand()/RAND_MAX;
    v[i]=vsc*(rn-0.5);
}
```

becomes e.g.:

```
//*** C++ ***

int mpiSize, mpiRank;

MPI_Comm_size(MPI_COMM_WORLD, &mpiSize);
MPI_Comm_rank(MPI_COMM_WORLD, &mpiRank);
int natLocal = nat / mpiSize;
double box = natLocal * mpiSize;
srand(time(NULL) + mpiRank); //Different seeds for different ranks
for (int i=0;i<natLocal;i++)
{
    x[i] = i + natLocal * mpiRank;
    rn=(double)rand()/RAND_MAX;
    v[i]=vsc*(rn-0.5);
}
```

In Fortran:

```
!*** Fortran ***

box=nat
x=[(real(i,rk),i=0,nat-1)]
call random_number(v)
v=vsc*(v-0.5)
```

becomes:

```
!*** Fortran ***

integer :: natLocal, mpiSize, mpiRank, mpiError
call mpi_comm_size(mpi_comm_world, mpiSize, mpiError)
call mpi_comm_rank(mpi_comm_world, mpiRank, mpiError)
natLocal = nat / mpiSize
box=natLocal * mpiSize
x=[(real(i + natLocal * mpiRank,rk),i=0,natLocal-1)]
call random_number(v)
v=vsc*(v-0.5)
```

Instead of hard-coding the periodic boundary condition, it is useful (though less elegant) to use "ghost" particles, so that (Fortran below):

```
//*** C++ ***

j=i-1; if (j<0) j=nat-1;
k=i+1; if (k>=nat) k=0;

dxl=x[i]-x[j];
dxr=x[k]-x[i];
```

becomes instead:

```
//*** C++ ***

std::vector<double> ghostX(2);
ghostX[0] = x.back();
ghostX[1] = x.front();
...
if ( i > 0 && i < nat-1)
{
    dxl=x[i]-x[i-1];
    dxr=x[i+1]-x[i];
}
else if ( i == 0 )
{
    dxl=x[i]-ghostX[0];
    dxr=x[i+1]-x[i];
}
else if ( i == nat - 1 )
{
    dxl=x[i]-x[i-1];
    dxr=ghostX[1]-x[i];
}
```

And in Fortran:

```
!*** Fortran ***

j=i-1; if (j<1) j=nat
k=i+1; if (k>nat) k=1

dxl=x(i)-x(j)
dxr=x(k)-x(i)
```

becomes:

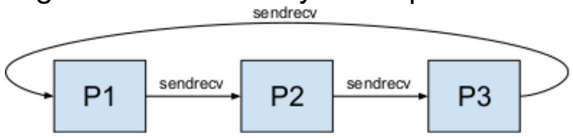
```
!*** Fortran ***

real(rk) :: ghostX(2)
ghostx(1) = x(nat)
ghostx(2) = x(1)
...
if ( i > 1 .and. i < nat ) then
    dxl = x(i) - x(i-1)
    dxr = x(i+1) - x(i)
else if ( i == 1 ) then
    dxl = x(i) - ghostX(0)
    dxr = x(i+1) - x(i)
else if ( i == nat ) then
    dxl = x(i) - x(i-1)
    dxr = ghostX(2) - x(i)
end if
```

This then makes it easy to parallelise. Instead of being set to the last and first local particles, `ghostX` will receive the position of the last particle of the previous rank, and the position of the first particle of the next rank (this information must be exchanged after each update of the positions):

```
//*** C++ ***

std::vector<MPI_Request> requests = std::vector<MPI_Request>(2);
std::vector<MPI_Status> statuses = std::vector<MPI_Status>(2);

MPI_Isend(&x.back(), 1, MPI_DOUBLE, nextRank, 0, comm, &requests[0]);
MPI_Recv(&ghostX.front(), 1, MPI_DOUBLE, previousRank, 0, comm,
   &statuses[0]);

MPI_Isend(&x.front(), 1, MPI_DOUBLE, previousRank, 1, comm, &requests[1]);
MPI_Recv(&ghostX.back(), 1, MPI_DOUBLE, nextRank, 1, comm, &statuses[1]);

MPI_Waitall(2, &requests[0], &statuses[0]);
```

```
!*** Fortran ***

type(MPI_Request) :: request(2)
type(MPI_Status) :: status(2)

call MPI_Isend(x(natLocal), 1, MPI_DOUBLE, nextRank, 0, &
                 comm, requests(1), rc)
call MPI_Recv(ghostX(1), 1, MPI_DOUBLE, previousRank, 0, &
                 comm, statuses(1), rc)

call MPI_Isend(x(1), 1, MPI_DOUBLE, previousRank, 1, &
                 comm, requests(2), rc)
call MPI_Recv(ghostX(2), 1, MPI_DOUBLE, nextRank, 1, &
                 comm, statuses(2), rc)

call MPI_Waitall(2,request,status,rc)
```

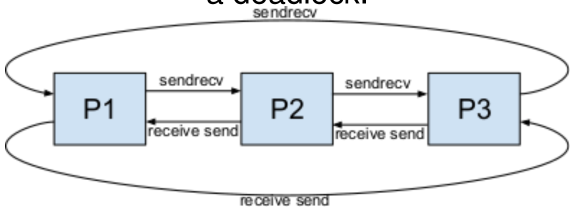
Note: Sendrecv may work with such a periodic topology when the amount of data is small, but it is more reliable to use non-blocking communications as above. This is an intended feature of MPI. For more information, look at MPI protocols.
Another way that sendrecv can be used is to flip the first side to receive send for each other process. Otherwise, the program most certainly will experience a deadlock.



Imagine the above example. This is the first time step, as all processes call sendrecv to the same side. In case the eager protocol has not been engaged, these are blocking in their receive parts. Therefore, we have the following dependency tree:

$$P1 \rightarrow P2 \rightarrow P3 \rightarrow P1$$

Meaning $P1$ waits for $P2$ to send, which in turns wait for $P3$ to send. Since $P3$ waits for $P1$ and since $P1$ will never send to $P3$ as its waiting for $P2$ to send it some message, we have a deadlock.



Above we see how a solution would look using blocking sendrecv.

One also needs to ensure that the code can generates the proper outputs. For the outputs of coordinates, each process should write to a different file:

```
//*** C++ ***

std::stringstream fileName;
fileName << "coords_" << mpiRank << ".dat";
coord_file = fileName.str();
```

In Fortran, `subroutine printcoords()` requires a bit more modifications, eg:

```
!*** Fortran ***

subroutine printcoords()
  integer :: ia, mpiRank, rc
  character(len=4)  :: mpiRank_char
  real(rk),parameter :: xsc=2.35
  call mpi_comm_rank(mpi_comm_world,mpiRank,rc)
  write(mpiRank_char,fmt='(i4.4)') mpiRank
  if (n==0) then !open a new file with unique id
     open(11,file='data_'//mpiRank_char//".txt", &
            action='write')
  else  !append to file with unique id
     open(11,file='data_'//mpiRank_char//".txt", &
            action='write',position='append')
  end if
  write(11,*) nat
  write(11,'(a,x,i0,x,i0,x,a,3f14.4)') 'Frame␣number␣',n,n,'␣fs␣
     boxsize',box,10.0,10.0
  do ia=1,nat
     write(11,'(a,x,4g20.10)') 'Fe',xsc*x(ia),0.0,0.0,ep(ia)
  end do
  close(11)
  return
end subroutine printcoords
```

And finally, we can use collective operations to calculate the total kinetic and potential energies on rank 0:

```
//*** C++ ***

epsum = std::accumulate(ep.begin(), ep.end(), 0.0);
eksum = std::accumulate(ek.begin(), ek.end(), 0.0);
if ( mpiInfo.rank == 0 )
{
    MPI_Reduce(MPI_IN_PLACE, &epsum, 1, MPI_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD);
    MPI_Reduce(MPI_IN_PLACE, &eksum, 1, MPI_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD);
}
else
{
    MPI_Reduce(&epsum, MPI_IN_PLACE, 1, MPI_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD);
    MPI_Reduce(&eksum, MPI_IN_PLACE, 1, MPI_DOUBLE, MPI_SUM, 0,
       MPI_COMM_WORLD);
}
```

```
!*** Fortran ***

epsum=sum(ep)
eksum=sum(ek)
if ( mpiRank == 0 )
{
    call MPI_Reduce(MPI_IN_PLACE, epsum, 1, MPI_DOUBLE, MPI_SUM, &
                    0, MPI_COMM_WORLD, rc)
    call MPI_Reduce(MPI_IN_PLACE, eksum, 1, MPI_DOUBLE, MPI_SUM, &
                    0, MPI_COMM_WORLD, rc)
}
else
{
    call MPI_Reduce(epsum, MPI_IN_PLACE, 1, MPI_DOUBLE, MPI_SUM, &
                    0, MPI_COMM_WORLD, rc)
    call MPI_Reduce(eksum, MPI_IN_PLACE, 1, MPI_DOUBLE, MPI_SUM, &
                    0, MPI_COMM_WORLD, rc)
}
```

The quality of our parallelisation can be measured in two common ways: so-called strong and weak scalings. In a strong scaling experiment, the system size is kept fixed, and the system is divided between the processes, so that each process has a fraction of the load. The ideal speedup $t(np = 1)/t(np)$ should be $\text{speedup} = np$, ie np processes should complete a task np times faster. In a weak scaling experiment, the system size is increased when the number of processes is increased, so that the load of each process (except communications) remains constant.Therefore the ideal speedup (or slowdown the inverse) should be 1 independent of np.

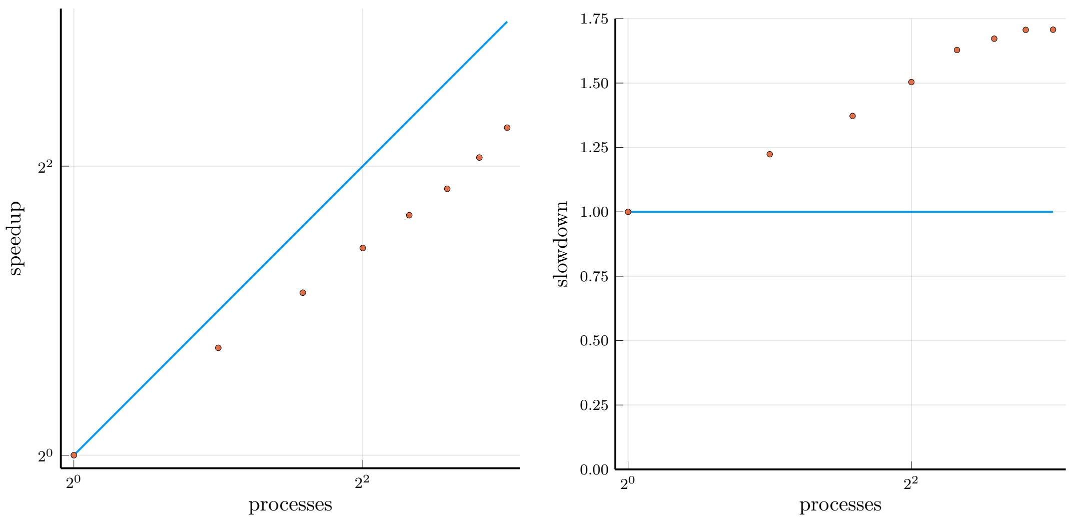


Figure 1