Summer School on PARALLEL COMPUTING

OpenMP Exercises

Marco Rorro - m.rorro@cineca.it

Neva Bešker - n.besker@cineca.it

CINECA - SuperComputing Applications and Innovation Department

CINECA

Warm-up with OpenMP

1. Compile and run ”Hello World” and experiment with the `OMP_NUM_THREADS` variable. If any errors occur, try to fix it.

2. Parallelize the MM (Matrix Multiplication) serial code acting only on the most important loop

Warm-up with OpenMP

1. Compile and run ”Hello World” and experiment with the `OMP_NUM_THREADS` variable. If any errors occur, try to fix it.
2. Parallelize the MM (Matrix Multiplication) serial code acting only on the most important loop

Hello World from C

C

```
#include <stdio.h>
#ifdef _OPENMP
#include<omp.h>
#endif
int main(int argc, char* argv[ ])
{
#ifdef _OPENMP
   int iam;
   #pragma omp parallel  /* the parallel block starts here */
   {
      iam=omp_get_thread_num();

      #pragma omp critical
         printf("Hello from %d\n",iam);

   } /* the parallel block ends here */
#else
   printf("Hello, this is a serial program.\n");
#endif
   return 0;
}
```

Hello World from Fortran

Fortran

```
Program Hello_from_Threads
#ifdef _OPENMP
   use omp_lib
#endif
   implicit none
   integer :: iam
#ifdef _OPENMP
   !$omp parallel
      iam=omp_get_thread_num()
      !$omp critical
         write, 'Hello from', iam
      !$omp end critical
   !$omp end parallel
#else
   write, 'Hello, this is a serial program'
#endif
end program Hello_from_Threads
```

Hello World from C

C

```
#include <stdio.h>
#ifdef _OPENMP
#include<omp.h>
#endif
int main(int argc, char* argv[ ])
{
#ifdef _OPENMP
   int iam;
   #pragma omp parallel \
   private(iam) /* the parallel block starts here */
   {
      iam=omp_get_thread_num();

      #pragma omp critical
         printf("Hello from %d\n",iam);

   } /* the parallel block ends here */
#else
   printf("Hello, this is a serial program.\n");
#endif
   return 0;
}
```

Hello World from Fortran

Fortran

```
Program Hello_from_Threads
#ifdef _OPENMP
   use omp_lib
#endif
   implicit none
   integer :: iam
#ifdef _OPENMP
   !$omp parallel &
   !$omp private(iam)
      iam=omp_get_thread_num()
      !$omp critical
         write, 'Hello from', iam
      !$omp end critical
   !$omp end parallel
#else
   write, 'Hello, this is a serial program'
#endif
end program Hello_from_Threads
```

Warm-up with OpenMP

1. Compile and run ”Hello World” and experiment with the `OMP_NUM_THREADS` variable. If any errors occur, try to fix it.

2. Parallelize the MM (Matrix Multiplication) serial code acting only on the most important loop

Matrix Multiplication in C

C

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc,char **argv) {
  int n;
  int i, j, k;
  ...
  double ( *a )[n] =  malloc(sizeof(double[n][n]));
  double ( *b )[n] =  malloc(sizeof(double[n][n]));
  double ( *c )[n] =  malloc(sizeof(double[n][n]));
  ...

  for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
      a[i][j] = ((double)rand())/((double)RAND_MAX);
      b[i][j] = ((double)rand())/((double)RAND_MAX);
      c[i][j] = 0.0;
  }


    for (i=0; i<n; ++i)
      for (k=0; k<n; k++)
        for (j=0; j<n; ++j)
          c[i][j] += a[i][k]*b[k][j];
  ...
  return 0;
}
```

Matrix Multiplication in C

C

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main(int argc,char **argv) {
  int n;
  int i, j, k;
  ...
  double ( *a )[n] =  malloc(sizeof(double[n][n]));
  double ( *b )[n] =  malloc(sizeof(double[n][n]));
  double ( *c )[n] =  malloc(sizeof(double[n][n]));
  ...

  for (i=0; i<n; i++)
    for (j=0; j<n; j++) {
      a[i][j] = ((double)rand())/((double)RAND_MAX);
      b[i][j] = ((double)rand())/((double)RAND_MAX);
      c[i][j] = 0.0;
  }

  #pragma omp parallel for private(j,k)
    for (i=0; i<n; ++i)
      for (k=0; k<n; k++)
        for (j=0; j<n; ++j)
          c[i][j] += a[i][k]*b[k][j];
  ...
  return 0;
}
```

Matrix Multiplication in Fortran

Fortran

```
program mat_prod
  implicit none
  integer :: n
  real(kind(1.d0)), dimension(:,:), allocatable :: a, b, c
  integer :: i, j, k
  ...
  allocate(a(n,n),b(n,n),c(n,n),stat=ierr)
  ...
  call random_number(a)
  call random_number(b)
  c = 0.d0


    do j=1, n
      do k=1, n
        do i=1, n
          c(i,j) = c(i,j) + a(i,k)*b(k,j)
        end do
      end do
    end do

  ...
end program mat_prod
```

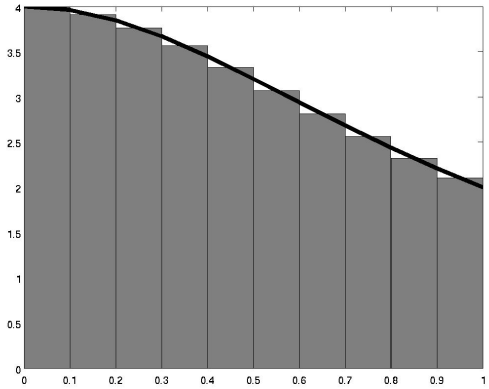
Matrix Multiplication in Fortran

Fortran

```
program mat_prod
  implicit none
  integer :: n
  real(kind(1.d0)), dimension(:,:), allocatable :: a, b, c
  integer :: i, j, k
  ...
  allocate(a(n,n),b(n,n),c(n,n),stat=ierr)
  ...
  call random_number(a)
  call random_number(b)
  c = 0.d0

  !$omp parallel do
    do j=1, n
      do k=1, n
        do i=1, n
          c(i,j) = c(i,j) + a(i,k)*b(k,j)
        end do
      end do
    end do
  !$omp end parallel do
  ...
end program mat_prod
```

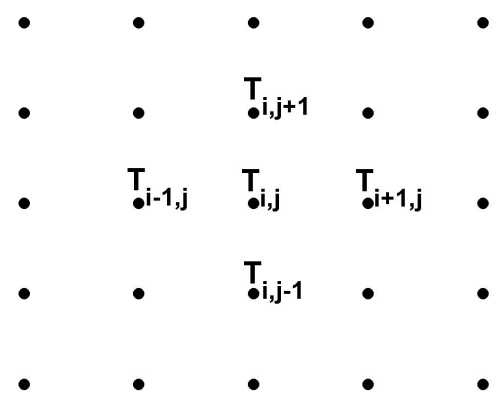
Let's play with OpenMP



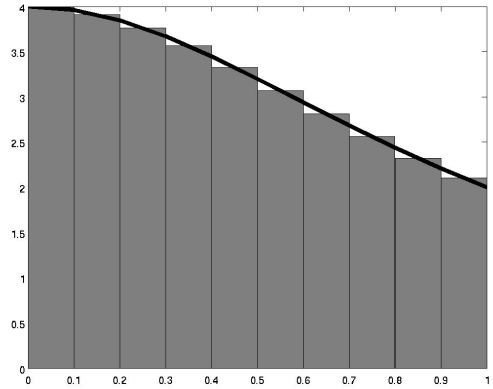
3. Parallelize the serial code **`Pi`**. It computes the Reimann approximation of
$\int_0^1 \frac{4}{1+x^2}\mathrm{d}x = 4\arctan x\Big|_0^1 = \pi$

4. Parallelize the serial code **`Laplace`**. It applies the iterative Jacobi method to a finite differences approximation of the Laplace equation with Dirichelet boundary condition:
$T_{i,j}^{n+1} = \frac{1}{4}(T_{i+1,j}^{n} + T_{i-1,j}^{n} + T_{i,j-1}^{n} + T_{i,j+1}^{n})$

   - start from the most computationally intensive loop
   - then try to include the **`while`** loop in the parallel region



Let's play with OpenMP



3. Parallelize the serial code `Pi`. It computes the Reimann approximation of
$\int_0^1 \frac{4}{1+x^2} dx = 4 \arctan x \Big|_0^1 = \pi$

4. Parallelize the serial code `Laplace`. It applies the iterative Jacobi method to a finite differences approximation of the Laplace equation with Dirichelet boundary condition:
$T_{i,j}^{n+1} = \frac{1}{4}(T_{i+1,j}^n + T_{i-1,j}^n + T_{i,j-1}^n + T_{i,j+1}^n)$
   - start from the most computationally intensive loop
   - then try to include the `while` loop in the parallel region

π

C

```
...

sum = 0.0;
dx = 1.0 / (double) intervals;

for (i = 1; i <= n; i++) {
   x = dx * ((double) (i - 0.5));
   f = 4.0 / (1.0 + x*x);
   sum = sum + f;
}
pi = dx*sum;

...
```

π

C

```
#include <omp.h>
...
double time1 = omp_get_wtime();
sum = 0.0;
dx = 1.0 / (double) intervals;
#pragma omp parallel for private(x,f) reduction(+:sum)
for (i = 1; i <= n; i++) {
   x = dx * ((double) (i - 0.5));
   f = 4.0 / (1.0 + x*x);
   sum = sum + f;
}
pi = dx*sum;
time2 = omp_get_wtime() - time1;
...
```

π

Fortran

```
...

sum=0.d0
dx=1.d0/intervals

do i=1,n
   x=dx*(i-0.5d0)
   f=4.d0/(1.d0+x*x)
   sum=sum+f
end do

pi=dx*sum

...
```

π

Fortran

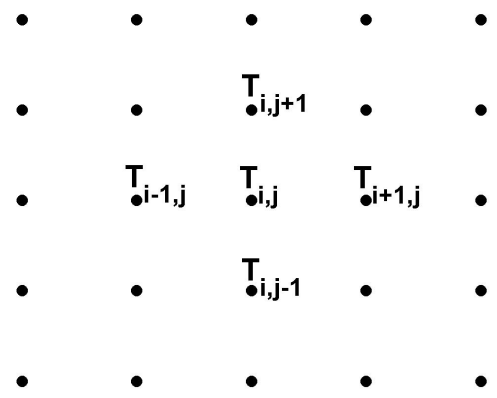
```
use omp_lib
...
time1 = omp_get_wtime()
sum=0.d0
dx=1.d0/intervals
!$omp parallel do private(x,f) reduction(+:sum)
do i=1,n
   x=dx*(i-0.5d0)
   f=4.d0/(1.d0+x*x)
   sum=sum+f
end do
!$omp end parallel do
pi=dx*sum
time2 = omp_get_wtime()
...
```

Let's play with OpenMP

3. Parallelize the serial code `Pi`. It computes the Reimann approximation of
$\int_0^1 \frac{4}{1+x^2} dx = 4 \arctan x \Big|_0^1 = \pi$

4. Parallelize the serial code **`Laplace`**. It applies the iterative Jacobi method to a finite differences approximation of the Laplace equation with Dirichelet boundary condition:
$T_{i,j}^{n+1} = \frac{1}{4}(T_{i+1,j}^{n} + T_{i-1,j}^{n} + T_{i,j-1}^{n} + T_{i,j+1}^{n})$

   - start from the most computationally intensive loop
   - then try to include the `while` loop in the parallel region



Laplace

C

```
...
while(var > tol && iter <= maxIter) {
  ++iter;
  var = 0.0;

  for (i=1; i<=n; ++i)
    for (j=1; j<=n; ++j) {
      Tnew[i*n2+j] = 0.25*(T[(i-1)*n2+j] + T[(i+1)*n2+j]
                           + T[i*n2+(j-1)] + T[i*n2+(j+1)]);
    var = fmax(var, fabs(Tnew[i*n2+j] - T[i*n2+j]));
  }
  Tmp=T; T=Tnew; Tnew=Tmp;
  if (iter%100 == 0)
    printf("iter: %8u, variation = %12.4lE\n", iter, var);
}
  ...
```

Laplace

C

```
...
while(var > tol && iter <= maxIter) {
  ++iter;
  var = 0.0;
  #pragma omp parallel for private(j) reduction(max:var)
  for (i=1; i<=n; ++i)
    for (j=1; j<=n; ++j) {
      Tnew[i*n2+j] = 0.25*(T[(i-1)*n2+j] + T[(i+1)*n2+j]
                           + T[i*n2+(j-1)] + T[i*n2+(j+1)]);
    var = fmax(var, fabs(Tnew[i*n2+j] - T[i*n2+j]));
  }
  Tmp=T; T=Tnew; Tnew=Tmp;
  if (iter%100 == 0)
    printf("iter: %8u, variation = %12.4lE\n", iter, var);
}
  ...
```

Reduction under the hood

C

```
while(var > tol && iter <= maxIter) {
  ++iter;
  var = 0.0;
  #pragma omp parallel
  {

    #pragma omp for private(j)
      for (i=1; i<=n; ++i)
        for (j=1; j<=n; ++j) {
          Tnew[i*n2+j] = 0.25*( T[(i-1)*n2+j] + T[(i+1)*n2+j]
                              + T[i*n2+(j-1)] + T[i*n2+(j+1)] );
          var  = fmax(var , fabs(Tnew[i*n2+j] - T[i*n2+j]));
        }


  }
  Tmp=T; T=Tnew; Tnew=Tmp;
  if (iter%100 == 0) ...
}
```

Reduction under the hood

C

```
while(var > tol && iter <= maxIter) {
  ++iter;
  var = 0.0;
  #pragma omp parallel
  {
    double pvar = 0.0;
    #pragma omp for private(j)
      for (i=1; i<=n; ++i)
        for (j=1; j<=n; ++j) {
          Tnew[i*n2+j] = 0.25*( T[(i-1)*n2+j] + T[(i+1)*n2+j]
                              + T[i*n2+(j-1)] + T[i*n2+(j+1)] );
          pvar  = fmax(pvar , fabs(Tnew[i*n2+j] - T[i*n2+j]));
        }
    #pragma omp critical
      if (pvar > var) var = pvar;
  }
  Tmp=T; T=Tnew; Tnew=Tmp;
  if (iter%100 == 0) ...
}
```

Laplace

Fortran

```
...
do while (var > tol .and. iter <= maxIter)
 iter = iter + 1
 var = 0.d0

 do j = 1, n
  do i = 1, n
   Tnew(i,j)=0.25d0*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))
   var = max(var, abs( Tnew(i,j) - T(i,j) ))
  end do
 end do

Tmp =>T; T =>Tnew; Tnew => Tmp;
if( mod(iter,100) == 0 ) ...
end do
...
```

Laplace

Fortran

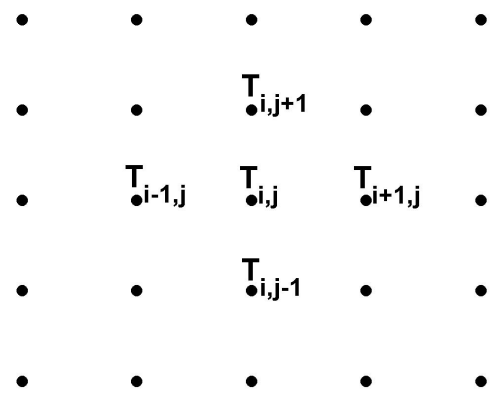
```
...
do while (var > tol .and. iter <= maxIter)
 iter = iter + 1
 var = 0.d0
 !$omp parallel do reduction(max:var)
 do j = 1, n
  do i = 1, n
   Tnew(i,j)=0.25d0*(T(i-1,j)+T(i+1,j)+T(i,j-1)+T(i,j+1))
   var = max(var, abs( Tnew(i,j) - T(i,j) ))
  end do
 end do
 !$omp end parallel do
Tmp =>T; T =>Tnew; Tnew => Tmp;
if( mod(iter,100) == 0 ) ...
end do
...
```

Let's play with OpenMP

3. Parallelize the serial code `Pi`. It computes the Reimann approximation of
$\int_0^1 \frac{4}{1+x^2} dx = 4 \arctan x \Big|_0^1 = \pi$

4. Parallelize the serial code **`Laplace`**. It applies the iterative Jacobi method to a finite differences approximation of the Laplace equation with Dirichelet boundary condition:
$T_{i,j}^{n+1} = \frac{1}{4}(T_{i+1,j}^{n} + T_{i-1,j}^{n} + T_{i,j-1}^{n} + T_{i,j+1}^{n})$

   - start from the most computationally intensive loop
   - then try to include the **`while`** loop in the parallel region



Laplace incremented

C

```
...

  while(var > tol && iter <= maxIter) {


      ++iter;
      var = 0.0;

    #pragma omp for private(j) reduction(max:var)
      ...


      Tmp=T; T=Tnew; Tnew=Tmp;
      if (iter%100 == 0) ...



  }

...
```

Laplace incremented

C

```
...
#pragma omp parallel
{
  while(var > tol && iter <= maxIter) {


      ++iter;
      var = 0.0;

    #pragma omp for private(j) reduction(max:var)
      ...


      Tmp=T; T=Tnew; Tnew=Tmp;
      if (iter%100 == 0) ...



  }
}
...
```

Laplace incremented

C

```
...
#pragma omp parallel
{
  while(var > tol && iter <= maxIter) {

    #pragma omp single
    {
      ++iter;
      var = 0.0;
    }
    #pragma omp for private(j) reduction(max:var)
      ...
    #pragma omp single
    {
      Tmp=T; T=Tnew; Tnew=Tmp;
      if (iter%100 == 0) ...

    }

  }
}
...
```

Laplace incremented

C

```
...
#pragma omp parallel
{
  while(var > tol && iter <= maxIter) {
    #pragma omp barrier
    #pragma omp single
    {
      ++iter;
      var = 0.0;

    #pragma omp for private(j) reduction(max:var)
      ...
    #pragma omp single nowait
    {
      Tmp=T; T=Tnew; Tnew=Tmp;
      if (iter%100 == 0) ...

    }

  }
}
...
```

Reduction under the hood

C

```
#pragma omp parallel
{
  while(var > tol && iter <= maxIter) {
    #pragma omp barrier
    #pragma omp single
    {
      ++iter;
      var = 0.0;
    }
    double pvar = 0.0;
    #pragma omp for nowait private(j)
      for (i=1; i<=n; ++i)
        for (j=1; j<=n; ++j) {
          Tnew[i*n2+j] = 0.25*( T[(i-1)*n2+j] + T[(i+1)*n2+j]
                              + T[i*n2+(j-1)] + T[i*n2+(j+1)] );
          pvar = fmax(pvar, fabs(Tnew[i*n2+j] - T[i*n2+j]));
        }
    #pragma omp critical
      if (pvar > var) var = pvar;

    #pragma omp barrier
    #pragma omp single nowait
    {
      Tmp=T; T=Tnew; Tnew=Tmp;
      if (iter%100 == 0) printf("iter: %8u, variation = %12.4lE\n", iter, var);
    }

  }
}
```

Laplace incremented

Fortran

```
...

  do while (var > tol .and. iter <= maxIter)


      iter = iter + 1
      var = 0.d0

  !$omp do reduction(max:var)
    ...
  !$omp end do

    Tmp =>T; T =>Tnew; Tnew => Tmp;
    if( mod(iter,100) == 0 ) ...

  end do

...
```

Laplace incremented

Fortran

```
...
!$omp parallel
  do while (var > tol .and. iter <= maxIter)


      iter = iter + 1
      var = 0.d0

  !$omp do reduction(max:var)
    ...
  !$omp end do

    Tmp =>T; T =>Tnew; Tnew => Tmp;
    if( mod(iter,100) == 0 ) ...

  end do
!$omp end parallel
...
```

Laplace incremented

Fortran

```
...
!$omp parallel
  do while (var > tol .and. iter <= maxIter)

  !$omp single
      iter = iter + 1
      var = 0.d0
  !$omp end single
  !$omp do reduction(max:var)
    ...
  !$omp end do
  !$omp single
    Tmp =>T; T =>Tnew; Tnew => Tmp;
    if( mod(iter,100) == 0 ) ...
  !$omp end single
  end do
!$omp end parallel
...
```

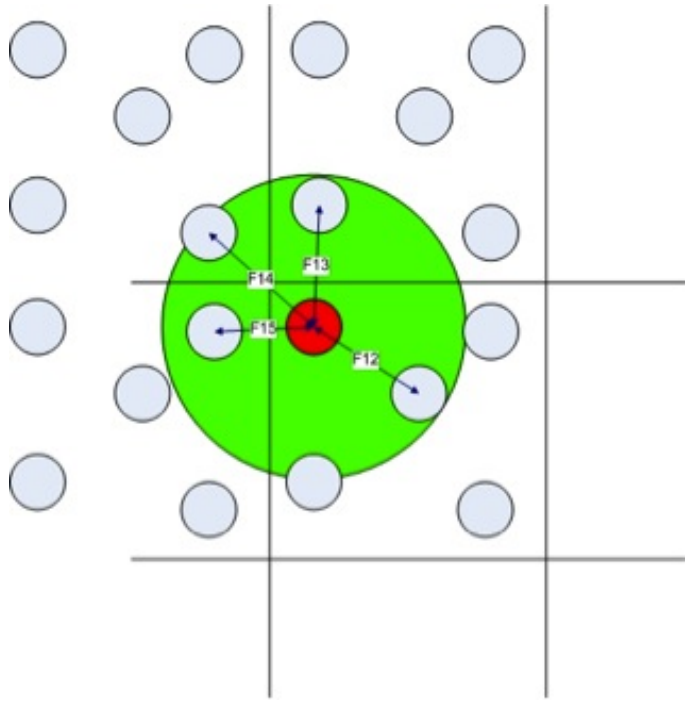
Laplace incremented

Fortran

```
...
!$omp parallel
  do while (var > tol .and. iter <= maxIter)
  !$omp barrier
  !$omp single
      iter = iter + 1
      var = 0.d0
  !$omp end single
  !$omp do reduction(max:var)
    ...
  !$omp end do
  !$omp single
    Tmp =>T; T =>Tnew; Tnew => Tmp;
    if( mod(iter,100) == 0 ) ...
  !$omp end single nowait
  end do
!$omp end parallel
...
```

When the Going Gets Tough, the Tough ...

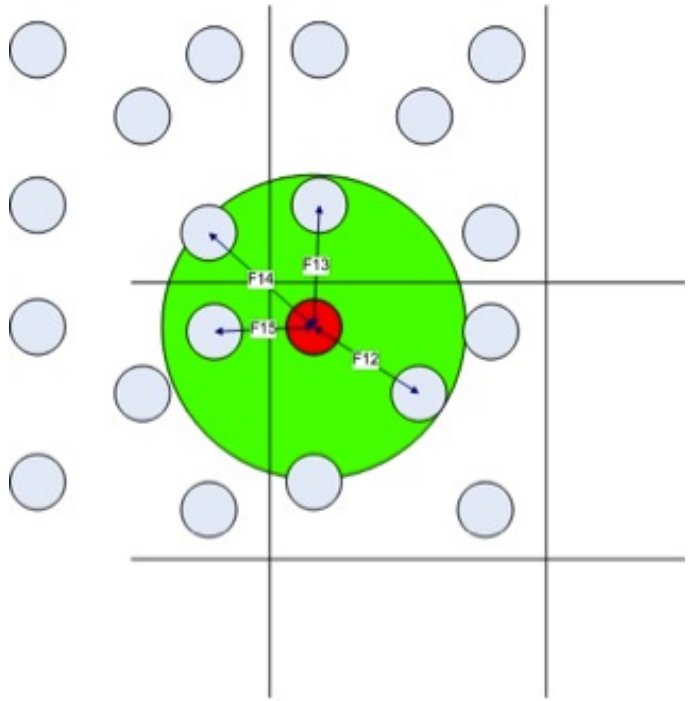
5. Parallelize the serial code **`Nbody`**. It computes the total energy and the forces of a system of N particles with potential $V = 1/r$ if r is less of a threshold and $V = 0$ otherwise.
   - pay attention to the update of **`forces`**
     - try to update them atomically
     - try to reduce them
   - try different schedules and test their performance



- to compile use the preprocessing MACRO DIM=55000, for example
  - **`gcc -O3 -DDIM=55000 Nbody.c -o nbody -lm`**

When the Going Gets Tough, the Tough ...

5. Parallelize the serial code **`Nbody`**. It computes the total energy and the forces of a system of N particles with potential $V = 1/r$ if r is less of a threshold and $V = 0$ otherwise.
   - pay attention to the update of **`forces`**
     - try to update them atomically
     - try to reduce them
   - try different schedules and test their performance



- to compile use the preprocessing MACRO DIM=55000, for example
  - `gcc -O3 -DDIM=55000 Nbody.c -o nbody -lm`

Nbody atomic

C

```
for(i=0; i<nbodies; ++i)
   for(j=i+1; j<nbodies; ++j) {
      d2 = 0.0;
      for(k=0; k<3; ++k) {
         rij[k] = pos[i][k] - pos[j][k];
         d2 += rij[k]*rij[k];
      }
      if (d2 <= cut2) {
         d = sqrt(d2);
         d3 = d*d2;
         for(k=0; k<3; ++k) {
            double f = -rij[k]/d3;

            forces[i][k] += f;

            forces[j][k] -= f;
         }
         ene += -1.0/d;
      }
   }
```

Nbody atomic

C

```
#pragma omp parallel for private(i,j,k,rij,d,d2,d3) reduction(+:ene)
   for(i=0; i<nbodies; ++i)
      for(j=i+1; j<nbodies; ++j) {
         d2 = 0.0;
         for(k=0; k<3; ++k) {
            rij[k] = pos[i][k] - pos[j][k];
            d2 += rij[k]*rij[k];
         }
         if (d2 <= cut2) {
            d = sqrt(d2);
            d3 = d*d2;
            for(k=0; k<3; ++k) {
               double f = -rij[k]/d3;

               forces[i][k] += f;

               forces[j][k] -= f;
            }
            ene += -1.0/d;
         }
      }
```

Nbody atomic

C

```
#pragma omp parallel for private(i,j,k,rij,d,d2,d3) reduction(+:ene)
   for(i=0; i<nbodies; ++i)
      for(j=i+1; j<nbodies; ++j) {
         d2 = 0.0;
         for(k=0; k<3; ++k) {
            rij[k] = pos[i][k] - pos[j][k];
            d2 += rij[k]*rij[k];
         }
         if (d2 <= cut2) {
            d = sqrt(d2);
            d3 = d*d2;
            for(k=0; k<3; ++k) {
               double f = -rij[k]/d3;
               #pragma omp atomic
               forces[i][k] += f;
               #pragma omp atomic
               forces[j][k] -= f;
            }
            ene += -1.0/d;
         }
      }
```

Nbody atomic

C

```
#pragma omp parallel for private(i,j,k,rij,d,d2,d3) reduction(+:ene) \
schedule(guided)
   for(i=0; i<nbodies; ++i)
      for(j=i+1; j<nbodies; ++j) {
         d2 = 0.0;
         for(k=0; k<3; ++k) {
            rij[k] = pos[i][k] - pos[j][k];
            d2 += rij[k]*rij[k];
         }
         if (d2 <= cut2) {
            d = sqrt(d2);
            d3 = d*d2;
            for(k=0; k<3; ++k) {
               double f = -rij[k]/d3;
               #pragma omp atomic
               forces[i][k] += f;
               #pragma omp atomic
               forces[j][k] -= f;
            }
            ene += -1.0/d;
         }
      }
```

Nbody atomic

Fortran

```
do i = 1, DIM
   do j = i+1, DIM
      rij(:) = pos(:,i) - pos(:,j)
      d2 = 0.d0
      do k = 1, 3
         d2 = d2 + rij(k)**2
      end do
      if (d2 .le. cut2) then
         d = sqrt(d2)
         f(:) = - 1.d0 / d**3 * rij(:)
         do k=1, 3

           forces(k,i) = forces(k,i) +  f(k)

           forces(k,j) = forces(k,j) -  f(k)
         end do
         ene = ene + (-1.d0/d)
      end if
   end do
end do
```

Nbody atomic

Fortran

```
!$omp parallel do private(i,j,k,rij,d,d2,f) reduction(+:ene)
   do i = 1, DIM
      do j = i+1, DIM
         rij(:) = pos(:,i) - pos(:,j)
         d2 = 0.d0
         do k = 1, 3
            d2 = d2 + rij(k)**2
         end do
         if (d2 .le. cut2) then
            d = sqrt(d2)
            f(:) = - 1.d0 / d**3 * rij(:)
            do k=1, 3

              forces(k,i) = forces(k,i) +  f(k)

              forces(k,j) = forces(k,j) -  f(k)
            end do
            ene = ene + (-1.d0/d)
         end if
      end do
   end do
!$omp end parallel do
```

Nbody atomic

Fortran

```
!$omp parallel do private(i,j,k,rij,d,d2,f) reduction(+:ene)
   do i = 1, DIM
      do j = i+1, DIM
         rij(:) = pos(:,i) - pos(:,j)
         d2 = 0.d0
         do k = 1, 3
            d2 = d2 + rij(k)**2
         end do
         if (d2 .le. cut2) then
            d = sqrt(d2)
            f(:) = - 1.d0 / d**3 * rij(:)
            do k=1, 3
              !$omp atomic
              forces(k,i) = forces(k,i) +  f(k)
              !$omp atomic
              forces(k,j) = forces(k,j) -  f(k)
            end do
            ene = ene + (-1.d0/d)
         end if
      end do
   end do
!$omp end parallel do
```

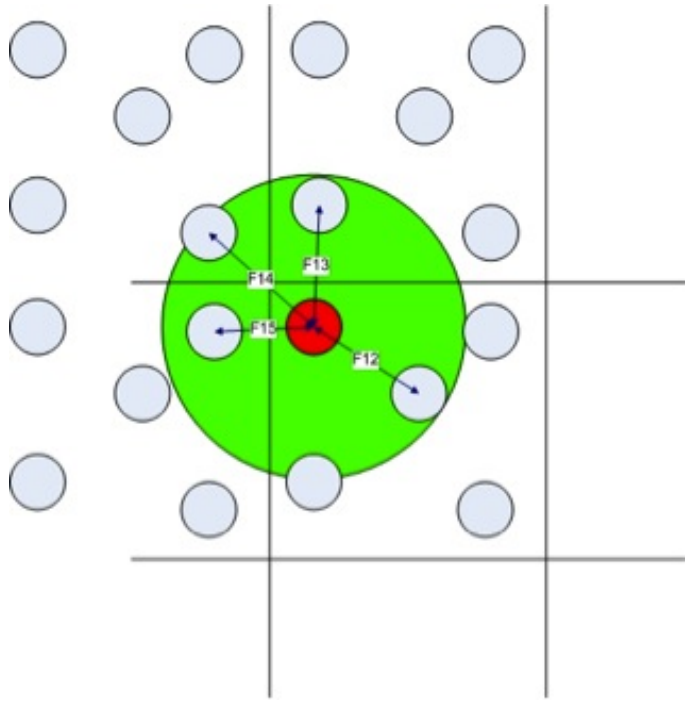
Nbody atomic

Fortran

```
!$omp parallel do private(i,j,k,rij,d,d2,f) reduction(+:ene) &
!$omp schedule(guided)
   do i = 1, DIM
      do j = i+1, DIM
         rij(:) = pos(:,i) - pos(:,j)
         d2 = 0.d0
         do k = 1, 3
            d2 = d2 + rij(k)**2
         end do
         if (d2 .le. cut2) then
            d = sqrt(d2)
            f(:) = - 1.d0 / d**3 * rij(:)
            do k=1, 3
              !$omp atomic
              forces(k,i) = forces(k,i) +  f(k)
              !$omp atomic
              forces(k,j) = forces(k,j) -  f(k)
            end do
            ene = ene + (-1.d0/d)
         end if
      end do
   end do
!$omp end parallel do
```

When the Going Gets Tough, the Tough ...

5. Parallelize the serial code **`Nbody`**. It computes the total energy and the forces of a system of N particles with potential $V = 1/r$ if r is less of a threshold and $V = 0$ otherwise.
   - pay attention to the update of **`forces`**
     - try to update them atomically
     - try to reduce them
   - try different schedules and test their performance



- to compile use the preprocessing MACRO DIM=55000, for example
  - `gcc -O3 -DDIM=55000 Nbody.c -o nbody -lm`

Nbody hand reduced in C

C ...

```
int tot_threads;
double ( *gforces )[3];

#pragma omp parallel private(i,j,k,rij,d,d2,d3)
{
#ifdef _OPENMP
   tot_threads = omp_get_num_threads();
#else
   tot_threads = 1;
#endif
   #pragma omp single
      gforces = calloc(nbodies*tot_threads,\
      sizeof( *gforces ));

      double ( *pforces )[3];

#ifdef _OPENMP
   pforces = gforces + nbodies*omp_get_thread_num();
#else
   pforces = gforces;
#endif
```

C

```
#pragma omp for reduction(+:ene) schedule(guided)
   for(i=0; i<nbodies; ++i)
      for(j=i+1; j<nbodies; ++j) {
         d2 = 0.0;
         for(k=0; k<3; ++k) {
            rij[k] = pos[i][k] - pos[j][k];
            d2 += rij[k]*rij[k];
         }
         if (d2 <= cut2) {
            d = sqrt(d2);
            d3 = d*d2;
            for(k=0; k<3; ++k) {
               double f = -rij[k]/d3;
               pforces[i][k] += f;
               pforces[j][k] -= f;
            }
            ene += -1.0/d;
         }
      }

   for(i=0; i<nbodies; ++i)
      for (j=0; j<tot_threads; j++)
         for(k=0; k<3; ++k)
            forces[i][k] += gforces[i+j*nbodies][k];
}
```

Nbody hand reduced in C

C ...

```
int tot_threads;
double ( *gforces )[3];

#pragma omp parallel private(i,j,k,rij,d,d2,d3)
{
#ifdef _OPENMP
   tot_threads = omp_get_num_threads();
#else
   tot_threads = 1;
#endif
   #pragma omp single
      gforces = calloc(nbodies*tot_threads,\
      sizeof( *gforces ));

      double ( *pforces )[3];

#ifdef _OPENMP
   pforces = gforces + nbodies*omp_get_thread_num();
#else
   pforces = gforces;
#endif
```

C

```
#pragma omp for reduction(+:ene) schedule(guided)
   for(i=0; i<nbodies; ++i)
      for(j=i+1; j<nbodies; ++j) {
         d2 = 0.0;
         for(k=0; k<3; ++k) {
            rij[k] = pos[i][k] - pos[j][k];
            d2 += rij[k]*rij[k];
         }
         if (d2 <= cut2) {
            d = sqrt(d2);
            d3 = d*d2;
            for(k=0; k<3; ++k) {
               double f = -rij[k]/d3;
               pforces[i][k] += f;
               pforces[j][k] -= f;
            }
            ene += -1.0/d;
         }
      }

   for(i=0; i<nbodies; ++i)
      for (j=0; j<tot_threads; j++)
         for(k=0; k<3; ++k)
            forces[i][k] += gforces[i+j*nbodies][k];
}
```

Nbody hand reduced in C

C ...

```
int tot_threads;
double ( *gforces )[3];

#pragma omp parallel private(i,j,k,rij,d,d2,d3)
{
#ifdef _OPENMP
   tot_threads = omp_get_num_threads();
#else
   tot_threads = 1;
#endif
   #pragma omp single
      gforces = calloc(nbodies*tot_threads,\
      sizeof( *gforces ));

      double ( *pforces )[3];

#ifdef _OPENMP
   pforces = gforces + nbodies*omp_get_thread_num();
#else
   pforces = gforces;
#endif
```

C

```
#pragma omp for reduction(+:ene) schedule(guided)
   for(i=0; i<nbodies; ++i)
      for(j=i+1; j<nbodies; ++j) {
         d2 = 0.0;
         for(k=0; k<3; ++k) {
            rij[k] = pos[i][k] - pos[j][k];
            d2 += rij[k]*rij[k];
         }
         if (d2 <= cut2) {
            d = sqrt(d2);
            d3 = d*d2;
            for(k=0; k<3; ++k) {
               double f = -rij[k]/d3;
               pforces[i][k] += f;
               pforces[j][k] -= f;
            }
            ene += -1.0/d;
         }
      }

   for(i=0; i<nbodies; ++i)
      for (j=0; j<tot_threads; j++)
         for(k=0; k<3; ++k)
            forces[i][k] += gforces[i+j*nbodies][k];
}
```

Nbody hand reduced in C

C ...

```
int tot_threads;
double ( *gforces )[3];

#pragma omp parallel private(i,j,k,rij,d,d2,d3)
{
#ifdef _OPENMP
   tot_threads = omp_get_num_threads();
#else
   tot_threads = 1;
#endif
   #pragma omp single
      gforces = calloc(nbodies*tot_threads,\
      sizeof( *gforces ));

      double ( *pforces )[3];

#ifdef _OPENMP
   pforces = gforces + nbodies*omp_get_thread_num();
#else
   pforces = gforces;
#endif
```

C

```
#pragma omp for reduction(+:ene) schedule(guided)
   for(i=0; i<nbodies; ++i)
      for(j=i+1; j<nbodies; ++j) {
         d2 = 0.0;
         for(k=0; k<3; ++k) {
            rij[k] = pos[i][k] - pos[j][k];
            d2 += rij[k]*rij[k];
         }
         if (d2 <= cut2) {
            d = sqrt(d2);
            d3 = d*d2;
            for(k=0; k<3; ++k) {
               double f = -rij[k]/d3;
               pforces[i][k] += f;
               pforces[j][k] -= f;
            }
            ene += -1.0/d;
         }
      }
#pragma omp for
   for(i=0; i<nbodies; ++i)
      for (j=0; j<tot_threads; j++)
         for(k=0; k<3; ++k)
            forces[i][k] += gforces[i+j*nbodies][k];
}
```

Nbody `reduction` in Fortran

Fortran

```
!$omp parallel do private(i,j,k,rij,d,d2,f) &
!$omp reduction(+:ene,forces) &
!$omp schedule(guided)
   do i = 1, DIM
      do j = i+1, DIM
         rij(:) = pos(:,i) - pos(:,j)
         d2 = 0.d0
         do k = 1, 3
            d2 = d2 + rij(k)**2
         end do
         if (d2 .le. cut2) then
         d = sqrt(d2)
            f(:) = - 1.d0 / d**3 * rij(:)
            forces(:,i) = forces(:,i) +  f(:)
            forces(:,j) = forces(:,j) -  f(:)
            ene = ene + (-1.d0/d)
         end if
      end do
   end do
!$omp end parallel do
```