Optimization
Avoid the avoidable

Luca Tornatore - I.N.A.F.

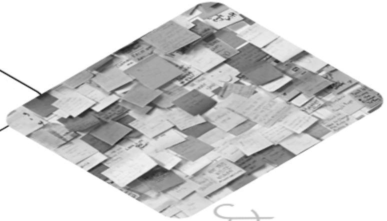
"Foundation of HPC" course

DATA SCIENCE &
SCIENTIFIC COMPUTING

2021-2022 @ Università di Trieste

Outline

First things first



Cache & Memory

Branches

Pipelines

Loops

Outline

Avoid the avoidable inefficiencies

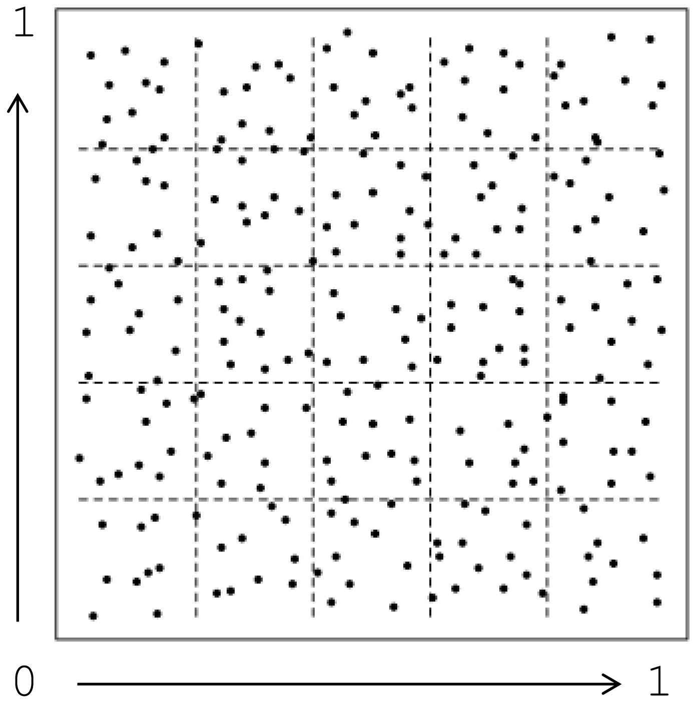
Branches

Pipelines

Loops

For the purpose of setting-up an example, let's suppose that

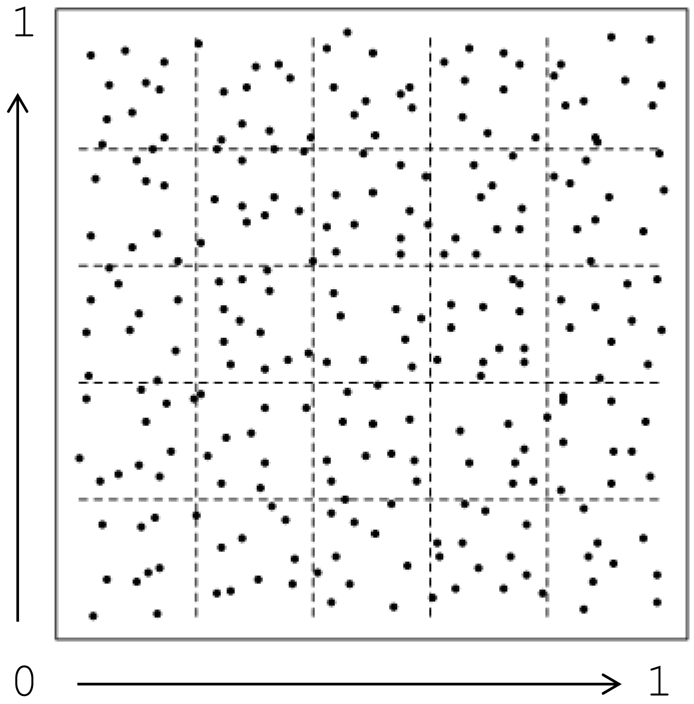
1) we have a distribution of random data points on a 2D plane which we subdivide in sub-regions using a grid.



For the purpose of setting-up an example, let's suppose that

1) we have a distribution of random data points on a 2D plane which we subdivide in sub-regions using a grid.

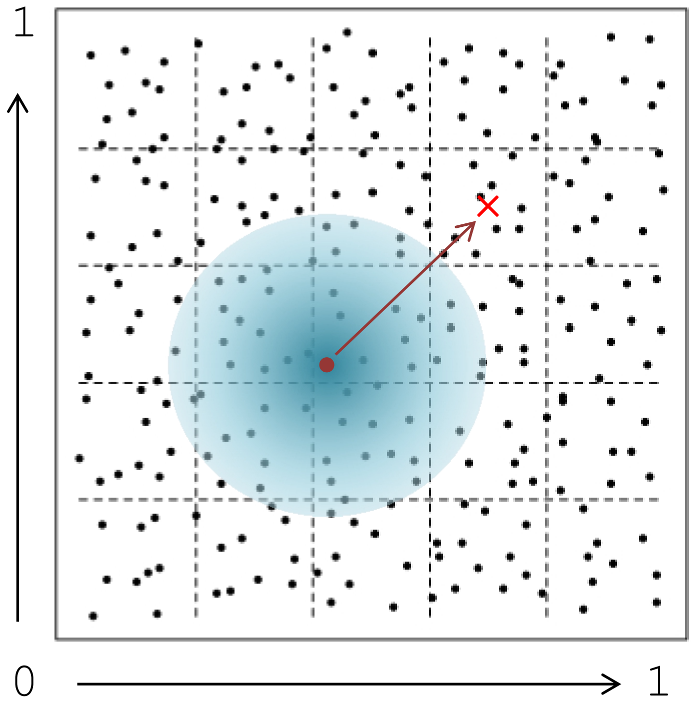
2) for each point *p*, we want to select all the grid cells whose center is closer to *p* than a given radius *r*, and to perform some operations accordingly to our search result.



For the purpose of setting-up an example, let's suppose that

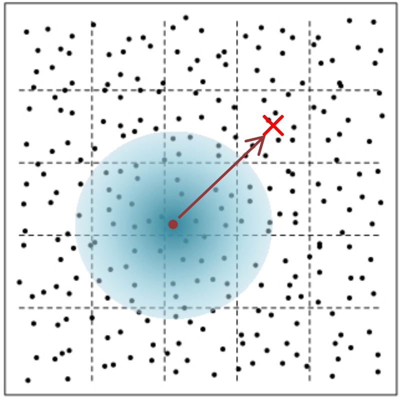
1) we have a distribution of random data points on a 2D plane which we subdivide in sub-regions using a grid.

2) for each point *p*, we want to select all the grid cells whose center is closer to *p* than a given radius *r*, and to perform some operations accordingly to our search result.



We may consider to use a nested loop like this one →

Is there anything you would change?



```
for(p = 0; p < Np; p++)

    for(i = 0; i < Ng; i++)
      for(j = 0; j < Ng; j++)
        for(k = 0; k < Ng; k++)
          {
            dist = sqrt(
                    pow(x[p] - (double)i/Ng - half_size, 2) +
                    pow(y[p] - (double)j/Ng - half_size, 2) +
                    pow(z[p] - (double)k/Ng - half_size, 2));

                 if(dist < R)
                    do something;
          }
```

SCO/avoid_the_avoidable/loop.0.c

| (1) Avoid expensive function calls

Some function calls are particularly expensive. Those include, among others, **`sqrt()`**, ...

Try to avoid them *if possible.*

```
for(p = 0; p < Np; p++)

    for(i = 0; i < Ng; i++)
      for(j = 0; j < Ng; j++)
        for(k = 0; k < Ng; k++)
          {
            dist2 = pow(x[p] - (double)i/Ng - half_size, 2) +
                    pow(y[p] - (double)j/Ng - half_size, 2) +
                    pow(z[p] - (double)k/Ng - half_size, 2));

                 if(dist2 < R2)
                    do something;
          }
```

SCO/avoid_the_avoidable/loop.1.c

(1) Avoid expensive function calls

Some function calls are particularly expensive. Those include, among others, **`sqrt()`**, **`pow()`**, ...

Try to avoid them *if possible.*

```
for(p = 0; p < Np; p++)

    for(i = 0; i < Ng; i++)
      for(j = 0; j < Ng; j++)
        for(k = 0; k < Ng; k++)
          {
            dx = x[p] - (double)i/Ng - half_size;
            dy = y[p] - (double)j/Ng - half_size;
            dz = z[p] - (double)k/Ng - half_size;

            dist2 = dx*dx + dy*dy + dz*dz;
            if(dist2 < R2)
               do something;
          }
```

SCO/avoid_the_avoidable/loop.2.c

(1) Avoid expensive function calls

Some function calls are particularly expensive. Those include, among others, **`sqrt()`**, **`pow()`**, **`floating point division`**, ..
Try to avoid them *if possible.*

```
for(p = 0; p < Np; p++)

    for(i = 0; i < Ng; i++)
      for(j = 0; j < Ng; j++)
        for(k = 0; k < Ng; k++)
          {
            dx = x[p] - (double)i * Ng_inv - half_size;
            dy = y[p] - (double)j * Ng_inv - half_size;
            dz = z[p] - (double)k * Ng_inv - half_size;

            dist2 = dx*dx + dy*dy + dz*dz;
            if(dist2 < R2)
               do something with sqrt(dist2);
          }
```

SCO/avoid_the_avoidable/loop.3.c

| (1) Avoid expensive function calls

```
(double)<i,j,k> * Ng_inv + half_size
```

was performed N^3+N^2+N times, always returning the same values.
Hoisting would save
$N(N^2+N^1+1)$ **`mul`**, **`add`** and **`mem`** accesses.

You can do better pre-computing the relevant values:

```
double ijk[Ng];
for(i = 0; i < Ng; i++)
   ijk[i] = i * Ng_inv + half_size
```

```
for(p = 0; p < Np; p++)

  for(i = 0; i < Ng; i++)
    for(j = 0; j < Ng; j++)
      for(k = 0; k < Ng; k++)
        {
          dx = x[p] - (double)i * Ng_inv - half_size;
          dy = y[p] - (double)j * Ng_inv - half_size;
          dz = z[p] - (double)k * Ng_inv - half_size;

          dist2 = dx*dx + dy*dy + dz*dz;
          if(dist2 < R2)
             do something with sqrt(dist2);
        }
```

| (2) *Hoisting* of expressions

`(double)<i,j,k> * Ng_inv + half_size`

was performed N^3+N^2+N times, always returning the same values.

Hoisting would save $N(N^2+N^1+1)$ `mul`, `add` and `mem` accesses.

```
for(i = 0; i < Ng; i++) {
   dx2 = x[p] - (double)i * Ng_inv - half_size;
   dx2 = dx2*dx2;

      for(j = 0; j < Ng; j++) {
         dy2 = y[p] - (double)j * Ng_inv - half_size;
         dy2 = dy2*dy2;
         dist2_xy = dx2 + dy2;

         for(k = 0; k < Ng; k++) {
            dz = z[p] - (double)k * Ng_inv - half_size;
            dist2 = dist2_xy + dz*dz;
            if(dist2 < Rmax2)
               do something with sqrt(dist2);  } } }
```

SCO/avoid_the_avoidable/loop.4.c

| (2) *Hoisting* of expressions

You could do even better by pre-computing the relevant values:

```
double ijk[Ng];
for(i = 0; i < Ng; i++)
   ijk[i] = i * Ng_inv + half_size
```

```
for(i = 0; i < Ng; i++) {
   dx2 = x[p] - Ng_inv[i] - half_size;
   dx2 = dx2*dx2;

      for(j = 0; j < Ng; j++) {
         dy2 = y[p] - Ng_inv[j] - half_size;
         dist2_xy = dx2 + dy2*dy2;

         for(k = 0; k < Ng; k++) {
            dz = z[p] - Ng_inv[k] - half_size;
            dist2 = dist2_xy + dz*dz;
            if(dist2 < Rmax2)
               do something with sqrt(dist2);  } } }
```

SCO/avoid_the_avoidable/loop.6.c

(3) Clarify the variables' scope

All these variables are very local, there's no need for them to have a wider scope.

That will help you in writing the code, and *may* help the compiler in optimizing the stack and perhaps the registers usage.

```
for(int i = 0; i < Ng; i++) {
   double dx2 = x[p] - (double)i * Ng_inv - half_size;
   dx2 *= dx2;

      for(j = 0; j < Ng; j++) {
         double dy2 = y[p] - (double)j * Ng_inv - half_size;
         double dist2_xy = dx2 + dy2*dy2;

         for(k = 0; k < Ng; k++) {
            double dz = z[p] - (double)k * Ng_inv - half_size;
            double dist2 = dist2_xy + dz*dz;

            if(dist2 < Rmax2)
               do something with sqrt(dist2);  } } }
```

| (4) Suggest what is important

These variables are often calculated and reused subsequently.

Keeping a register dedicated to them may be useful.

Note: *this is a suggestion, the compiler, after analyzing the code, may decide differently*

```
double register Ng_inv = 1.0 / Ng;
for(int i = 0; i < Ng; i++) {
   double dx2 = x[p] - (double)i * Ng_inv - half_size;
   dx2 *= dx2;

      for(j = 0; j < Ng; j++) {
         double dy2 = y[p] - (double)j * Ng_inv - half_size;
         dy2 *= dy2;
         double register dist2_xy = dx2 + dy2;

         for(k = 0; k < Ng; k++) {
            double register dz = z[p] - (double)k * Ng_inv - …;
            double register dist2 = dist2_xy + dz*dz;

            if(dist2 < Rmax2)
               do something with sqrt(dist2);  } } }
```

SCO/avoid_the_avoidable/loop.5.c

(5) Don't repeat unnecessary checks

Do you expect any great performance from this code ?

If not, why ?

```
char * find_char_in_string( char *string, char c )
{
    int i = 0;
    while ( i < strlen(string) )
       if( string[i] == c )
         break;
       else
         i++;

    if( i < strlen(string) )
       return &string[i];
    else
       return NULL;
}
```

(5) Don’t repeat unnecessary checks

There are several details that dump the performance, i.e. the CPE, of this loop.

The one I want to draw your attention to is the repeated call to the `strlen()` function.

Do you expect the string to change while you are scanning it?
I guess no, but the compiler does not know it and has no way to understand that by code analysis.
Moreover, the memory pointed by string could be modified somewhere else between two iterations.

```
char * find_char_in_string( char *string, char c )
{
    int i = 0;
    while ( i < strlen(string) )
       if( string[i] == c )
         break;
       else
         i++;

    if( i < strlen(string) )
       return &string[i];
    else
       return NULL;
}
```

(5) Don't repeat unnecessary checks

This very simple change will save you a lot of CPE

```
char * find_char_in_string( char *string, char c )
 {
    int i = 0;
    int len = strlen(string);
    while ( i < len )
       if( string[i] == c )
         break;
       else
         i++;

    if( i < strlen(string) )
       return &string[i];
    else
       return NULL;
 }
```

(5) Don’t repeat unnecessary checks

For a number of reasons, this version is even more efficient than the previous one.

Can you tell why ?

```
char * find_char_in_string( char *string, char c )
 {
    char *pos = string;
    while( ( *pos != ‘\0’ ) && ( *pos != c ) )
       pos++;

    return ( *pos == ‘\0’ ? NULL : pos );
 }
```

(6) Avoid unnecessary memory references

This simple loop for a reduction of an array accumulates the partial results de-referencing the pointer `sum` at each iteration.

```
void reduce_vector( int n, double *array, double *sum )
 {
    for ( int i = 0; i < n; i++ )
         *sum += array[i];
    return;
 }
```

(6) Avoid unnecessary memory references

This simple loop for a reduction of an array accumulates the partial results de-referencing the pointer `sum` multiple times.

(asm obtained with -O1)

```
void reduce_vector( int n, double *array, double
*sum )
 {
    for ( int i = 0; i < n; i++ )
          *sum += array[i];
    return;
 }
```

```
.L3:
  movsd xmm0, QWORD PTR [rdx]
  addsd xmm0, QWORD PTR [rax]
  movsd QWORD PTR [rdx], xmm0
  add   rax, 8
  cmp   rax, rcx
  jne   .L3
```

```
movsd   xmm0, value of *sum
addsd   xmm0, value of *array
movsd   address of sum, xmm0
add     rax, 8           ( array++ )
cmp     rax, n
```

(6) Avoid unnecessary memory references

Introducing a separated, local accumulator will save memory accesses

(asm obtained with -O1)

```
void reduce_vector( int n, double *array, double
*sum )
 {
    double cum = 0;
    for ( int i = 0; i < n; i++ ) cum += array[i];
    *sum = cum;
    return;
 }
```

```
.L11:
  addsd xmm0, QWORD PTR [rax]
  add   rax, 8
  cmp   rax, rdx
  jne   .L11
```

```
addsd     xmm0, value of *array
add       rax, 8       (array++)
cmp       rax, rdx     (array with end-of-array)
```