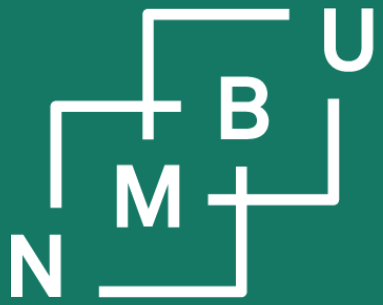
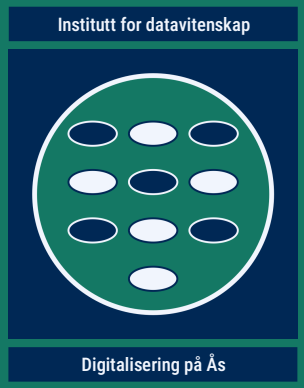


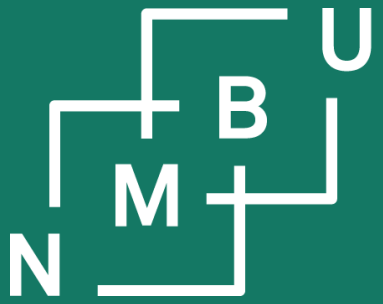
Norges miljø- og biovitenskapelige universitet



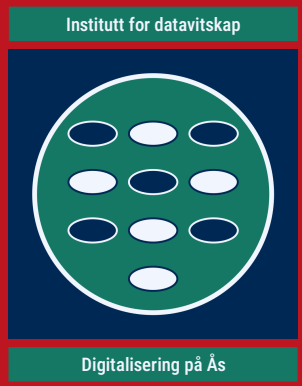
INF205
Resource-efficient programming

4 Production

4.5 "#define" macros
4.6 Shared libraries
4.7 Benchmarks
4.8 CMake



Noregs miljø- og biovitskaplege universitet



4 Production

4.5 "#define" macros

INF205

16th November 2022

“#define” macros

We have been using the preprocessor directive “**#define**” to protect headers:

```
#ifndef HEADER_NAME_H
#define HEADER_NAME_H

… // content of the header

#endif
```

This particular use is still recommended.

Other uses of **#define** are seen very frequently in legacy code and projects that have existed for a long time … such as many libraries!

The C/C++ preprocessor accepts defines both as directives or, equivalently, by define flags passed through the compiler, *e.g.*:

```
g++ -DNDEBUG -DPARAM=7 -c -o object_file.o code_file.cpp
```

The “NDEBUG” define also deactivates “assert” and a few other constructions.

Case distinctions using macros

C/C++ preprocessor directives are a programming language in its own right.

```
#ifdef SOME_CONSTANT
  // here comes some code
#else
  // alternative solution
#endif
```

Exercise

Use macros for a sequential version of the MPI-parallel primes counting code from the same **single code base**.

Overusing preprocessor directives makes code *hard to debug and maintain*!

Features that are turned on through -D… *directives can interact* in unforeseen ways. If there are *n options*, we would need to test 2^n *versions* of the code.

If it does not cost much in performance, make **case distinctions at runtime**.

Where possible, **better use a template** rather than a #define. In the future, **generic programming** will make template-like constructions more powerful.

Case distinctions using macros

```
…
#ifdef USE_MPI
#include <mpi.h>
#endif
…
int main(int argc, char** argv)
{
  …
#ifdef USE_MPI
  MPI_Init(&argc, &argv);

  int size = 0;
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  int rank = 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
#else
  int size = 1;
  int rank = 0;
#endif
```

```
  int64_t limit = std::atoi(argv[1]);
  …
  int64_t counted_primes = 0;

  for(int64_t n = 6*(rank+1) - 1; n < limit; n += 6*size)
    if(is_prime(n)) counted_primes++;

  for(int64_t n = 6*(rank+1) + 1; n < limit; n += 6*size)
    if(is_prime(n)) counted_primes++;

  int64_t overall_primes = 0;

#ifdef USE_MPI
  MPI_Reduce(&counted_primes, &overall_primes, …);
  if(!rank) overall_primes += 2;
  MPI_Finalize();
#else
  overall_primes = counted_primes + 2;
#endif
  …
}
```

Style advice on #define macros

#define NAME	is equivalent to	g++ -DNAME
#define NAME value	is equivalent to	g++ -DNAME=value

Core guidelines about defines:

- SF.8: Use #include guards for all header files
- ES.30: Don't use macros for program text manipulation
- ES.31: Don't use macros for constants or "functions"

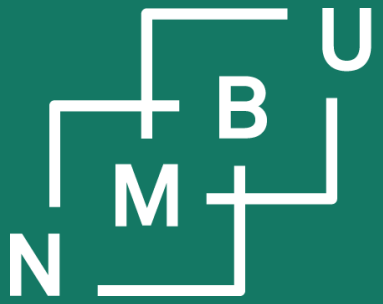
```
#define PI 3.14
#define SQUARE(a, b) (a * b)
```

("Example, bad", from Core Guidelines)

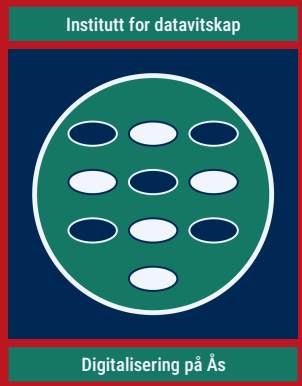
- ES.32: Use ALL_CAPS for all macro names
- ES.33: If you must use macros, give them unique names

```
#define MYCHAR       /* BAD, will eventually clash with someone else's MYCHAR*/
#define ZCORP_CHAR   /* Still evil, but less likely to clash */
```

(from Core Guidelines)



Noregs miljø- og biovitskaplege universitet



4 Production

4.5 "#define" macros

4.6 Shared libraries

Shared libraries

In programming with C/C++ libraries, we have already seen:

- How to work with the C standard library, *e.g.*, <cassert>, <cmath>, …
- … with the C++ standard library, with the STL, OpenMPI, and ROS

Technically, libraries are pre-compiled object code that can be reused.

The library needs to be accessed at three stages:

- At **compile time**, we need to **include** the **library headers**.
  - The complete source code for the libraries is unnecessary.
    It is even possible for the library to be coded in another language.
- During **linking**, the object code is **dynamically linked** against the library.
  - At this stage, the library "**static object**" (*.so file) is needed.
  - The executable does not contain the library's object code!
- At **execution time**, the executable and the library are **loaded jointly**.
  - If the library's **static object** code is gone now, the code will not run!

Library use example

The **image-benchmark** code uses the Magick++ API of ImageMagick.[1, 2]

```
#include <Magick++.h>

int main(int argc, char** argv)
{
  // charmap input
  std::ifstream pixistrm(argv[1]);
  diskgraphics::Charmap cm;
  pixistrm >> cm;
  pixistrm.close();

  // image object setup
  Magick::InitializeMagick(*argv);
  Magick::Image img(Magick::Geometry(cm.get_sizex(), cm.get_sizey()), "white");
  img.magick("BMP");
  img.monochrome();
  img.type(Magick::BilevelType);

  // pixel-by-pixel transfer of content
  for(int x = 0; x < cm.get_sizex(); x++)
    for(int y = 0; y < cm.get_sizey(); y++)
      img.pixelColor(x, y, Magick::Color(cm.get_pixel(x, y) == 0? "black": "white"));

  // output in BMP format using one bit per pixel
  img.quantize(2);
  img.write(argv[2]);
}
```

read a "Charmap" object from a pixel graphics file, using last week's file format

ImageMagick can deal with many file formats; we need an uncompressed pixel graphics format such as BMP

copy colour value of pixels

with quantize(2) we get one bit per pixel

[1]Magick++ API documentation: https://imagemagick.org/Magick++/Documentation.html

[2]Magick++ Tutorial: https://www.imagemagick.org/Magick++/tutorial/Magick++_tutorial.pdf

Library use: Compiling and linking

The **image-benchmark** code uses the Magick++ API of ImageMagick.[1, 2]

```
#include <Magick++.h>
```

How does the compiler know where to look for this file?

For this particular library, there is a tool that helps call g++ with the right flags:

```
g++ -c -std=c++17 -o <name>.o <name>.cpp `Magick++-config --cppflags`
```

```
-fopenmp -DMAGICKCORE_HDRI_ENABLE=1
-DMAGICKCORE_QUANTUM_DEPTH=16 -I/usr/local/include/ImageMagick-7
```

```
g++ -std=c++17 -o <name> *.o `Magick++-config --libs`
```

```
-L/usr/local/lib -lMagick++-7.Q16HDRI -lMagickWand-7.Q16HDRI -lMagickCore-7.Q16HDRI
```

More typically, you need to provide this information to the compiler by hand.

[1]Magick++ API documentation: https://imagemagick.org/Magick++/Documentation.html

[2]Magick++ Tutorial: https://www.imagemagick.org/Magick++/tutorial/Magick++_tutorial.pdf

Library programming

A shared object can be created from an object file using g++ **-shared**:

```
g++ -c -o first.o first.cpp
g++ -c -o second.o second.cpp
g++ -shared -o libname.so first.o second.o
```

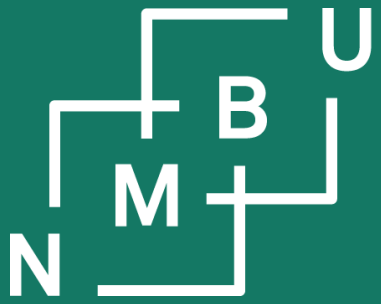
this is a capital i, not a lower-case L

The library header location can be passed to g++ at compile time with -I…, and the shared object is found by the linker with the -L and -l options.

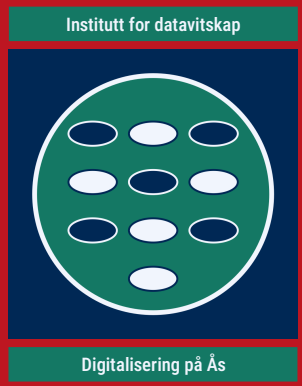
this time it is a lower-case L

But the library also needs to be found at execution time. For that to work, it must be in the appropriate path, or one of the environment variables for library paths must be set to include the location of the shared object.

this can be $LD_LIBRARY_PATH



Noregs miljø- og biovitskaplege universitet



4 Production

4.5 "#define" macros

4.6 Shared libraries

4.7 Benchmarks

Role of the benchmarks

What are our benchmarks like?

- The benchmark is not a single input file. It is an **input file generator**.
- The **problem size** is controlled through one or multiple parameters.

What to do with the benchmark:

- Record **average runtimes**, maybe worst-case runtimes, on a single core
- Use it to conduct **weak scaling** tests in parallel
  (weak scaling: the problem size increases with the number of cores)

What *not* to do with the benchmark:

- The program should not *assume* that it only receives benchmark input: It should work for the problem *in general*.
- The benchmark is there so you can document how your code performs and scales. If you tailor the code so specifically to the benchmark that it becomes worse (or incorrect) overall, it has failed its purpose.

Benchmark scenario: Disk graphics

Norwegian University of Life Sciences

See **image-benchmark** generator code. Main scenario parameter:

- Graphics edge size *a*, in pixels, so that the figure has a^2 pixels

Additional benchmark scenario parameter (change only if you have a reason):

- number of disks *m* from which the figures are generated; default: *m* = 10

```
float disk_radius = a;
unsigned char disk_colour = 0;
for(int i = 0; i < m; i++)  {
  diskgraphics::Disk dsk;
  disk_radius /= 1.125 + 0.125*rand()/(float)RAND_MAX;

  dsk.x = a * (0.75*(rand()/(float)RAND_MAX) + 0.125);
  dsk.y = a * (0.75*(rand()/(float)RAND_MAX) + 0.125);
  dsk.r = disk_radius;
  dsk.colour = disk_colour;
  dv.add_disk(dsk);

  if(std::rand() > RAND_MAX/3.0) disk_colour = 255 - disk_colour;
}
```

./generator (from **generator.cpp**) creates a disk vector with m disks and the figure in charmap format.

./bitmap (from **bitmap.cpp**) uses Magick++ from ImageMagick to convert charmaps to BMP format.

Norwegian University of Life Sciences

Benchmark scenario: Disk graphics

Terminal output of **./generator 32 && ./bitmap**

```
Image edge size:        32 pixels (32 x 32)
No. random disks:       10
Pixel output to:        benchmark.pxl
Vector output to:       benchmark.vct
32 32
...
-rw-r--r-- 1 horsch ch     258 nov.  15 13:26 benchmark.bmp
-rw-r--r-- 1 horsch ch    1030 nov.  15 13:26 benchmark.pxl
-rw-r--r-- 1 horsch ch     282 nov.  15 13:26 benchmark.vct
...
32 32
...
horsch@raviolixt:/arc/tr/lehre/2022/inf205/4-production/examples/image-benchmark$
```

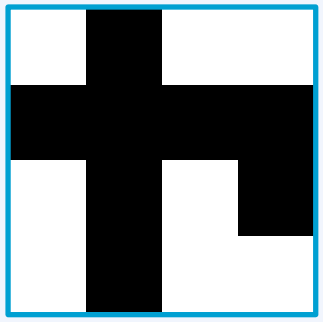
BMP and VCT files about the same size.

benchmark.bmp file

Benchmark scenario: Disk graphics

A series of figures is uploaded as **disk-benchmark-series.zip**, including these:

4x4 pixels



146 byte BMP

16x16 pixels

192 byte BMP

64x64 pixels

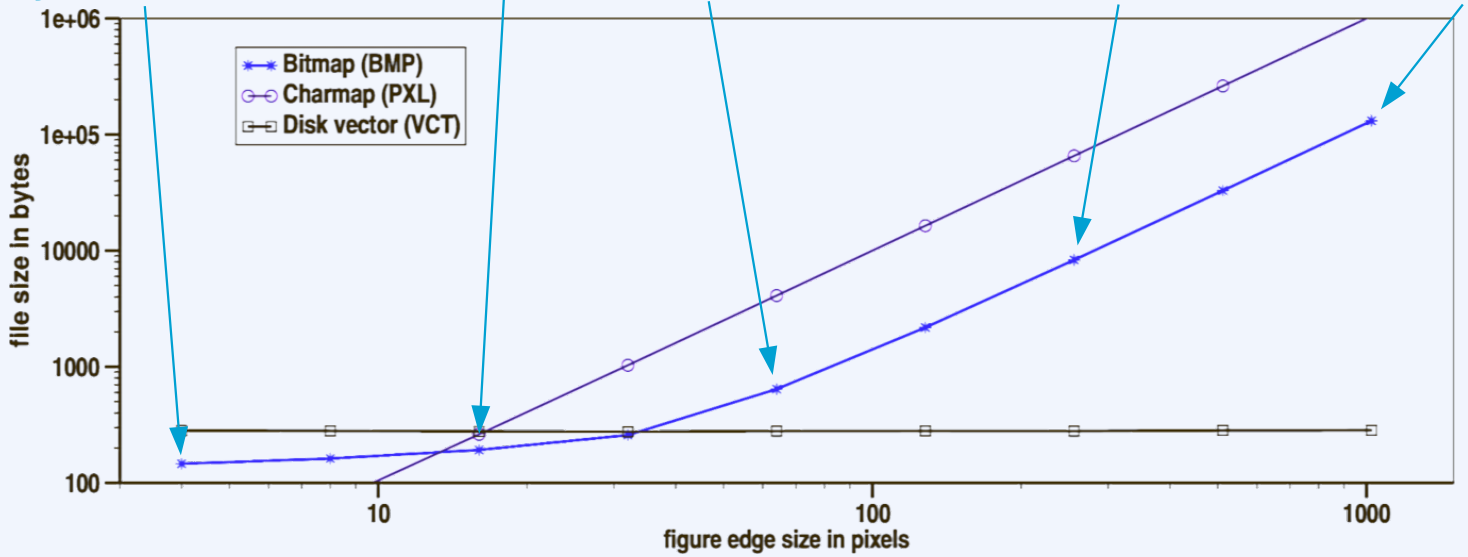
642 byte BMP

256x256 pixels

8.1 KB BMP

1024x1024 pixels

128 KB BMP



Benchmark scenario: ANN

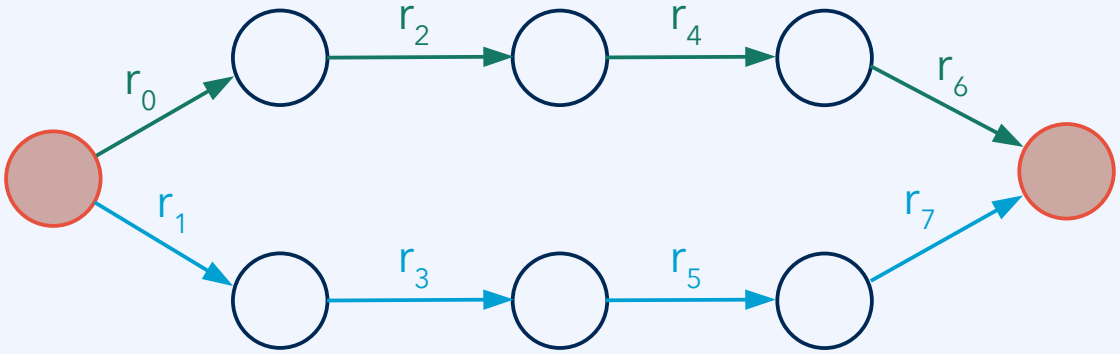
Use the ANN as a compressed representation of multiple benchmark figures.

Benchmark scenario: Graph queries

See **graph-benchmark** generator code. Two main scenario parameters:

- Graph size *n*, the number of nodes in the graph (not number of edges)
- Query size *m*, length of each of the two paths from the query

Benchmark query pattern for $m = 4$



query.dat output file

```
<r0> <r2> <r4> <r6> <>
<r1> <r3> <r5> <r7> <>
```

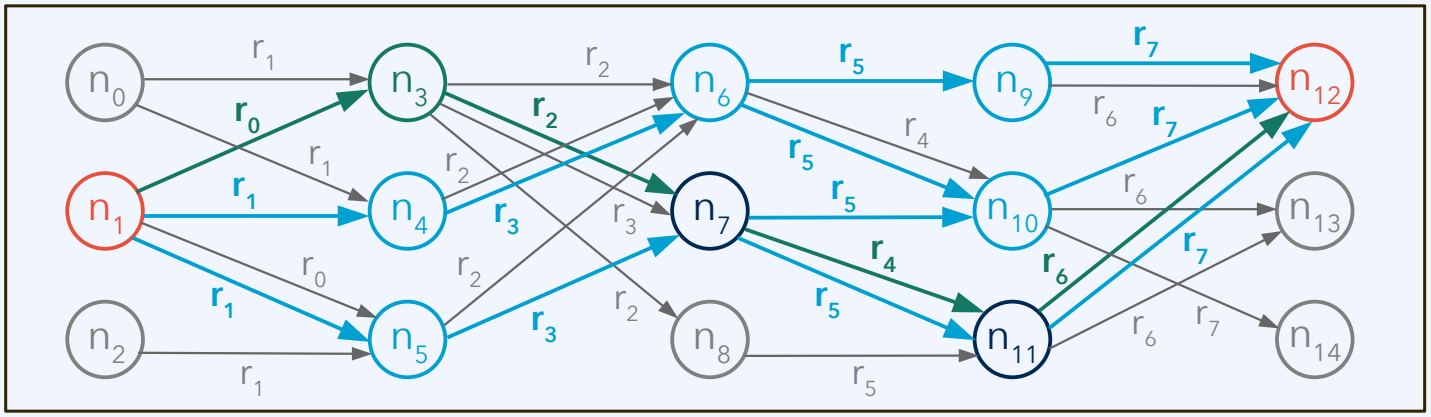
$p = \{r_0, r_2, r_4, r_6\}$

$q = \{r_1, r_3, r_5, r_7\}$

Random graphs are generated such that both cases with and without an instance of the graph pattern occur (the aim was to approximate a 50:50 ratio).

Benchmark scenario: Graph queries

Random benchmark graph from "./generate-graph 15 4 kb.dat query.dat":



<n0>	<r1>	<n3>	<n3>	<r2>	<n6>	**<n6>**	**<r5>**	**<n9>**	<n9>	<r6>	<n12>	<>
<n0>	<r1>	<n4>	**<n3>**	**<r2>**	**<n7>**	<n6>	<r4>	<n10>	**<n9>**	**<r7>**	**<n12>**	
<n1>	**<r0>**	**<n3>**	<n3>	<r3>	<n7>	**<n6>**	**<r5>**	**<n10>**	**<n10>**	**<r7>**	**<n12>**	
<n1>	**<r1>**	**<n4>**	<n3>	<r2>	<n8>	**<n7>**	**<r5>**	**<n10>**	<n10>	<r6>	<n13>	
<n1>	<r0>	<n5>	<n4>	<r2>	<n6>	**<n7>**	**<r4>**	**<n11>**	<n10>	<r7>	<n14>	
<n1>	**<r1>**	**<n5>**	**<n4>**	**<r3>**	**<n6>**	**<n7>**	**<r5>**	**<n11>**	**<n11>**	**<r6>**	**<n12>**	
<n2>	<r1>	<n5>	<n5>	<r2>	<n6>	<n8>	<r5>	<n11>	**<n11>**	**<r7>**	**<n12>**	
			<n5>	**<r3>**	**<n7>**				<n11>	<r6>	<n13>	

kb.dat
output file

The benchmark graphs consist of $m+1$ partitions. Assuming that n and m grow in proportion, the number of edges is in $O(n)$, generating sparse graphs.

Benchmark scenario: Sphere overlap

Norwegian University of Life Sciences

See **sphere-benchmark** generator code. Main scenario parameter:

- *N*, the number of spherical particles in the system

Additional benchmark scenario parameters (change only if you have a reason):

- packing fraction ξ, *i.e.*, total sphere volume / box volume; default: $\xi = 7/9$
- ratio ζ_{max} between largest and smallest sphere diameter; default: $\zeta = 10/3$

Remark: If the spheres are all of the same size, the densest packing (without any overlaps) has the packing fraction 0.7405.

This had been known as one of the "Hilbert problems."

THEOREM 1.1 (The Kepler conjecture). *No packing of congruent balls in Euclidean three space has density greater than that of the face-centered cubic packing.*

This density is $\pi/\sqrt{18} \approx 0.74$.

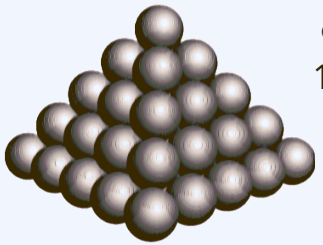


Figure 1.1: The face-centered cubic packing

The proof of this result is presented in this paper. Here, we describe the

T. C. Hales, "A proof of the Kepler conjecture," *Ann. Math.* **162**(3): 1065–1185, doi:10.4007/annals.2005.162.1065, **2005**.

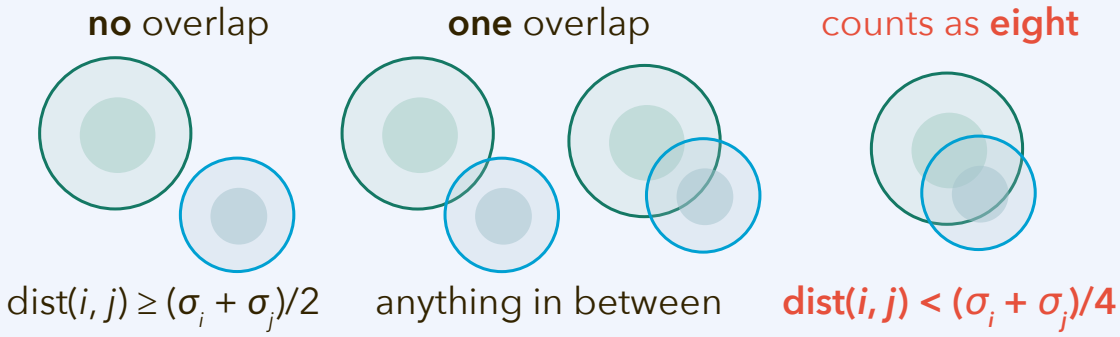
Benchmark scenario: Sphere overlap

Norwegian University of Life Sciences

example output from **./generator 256**

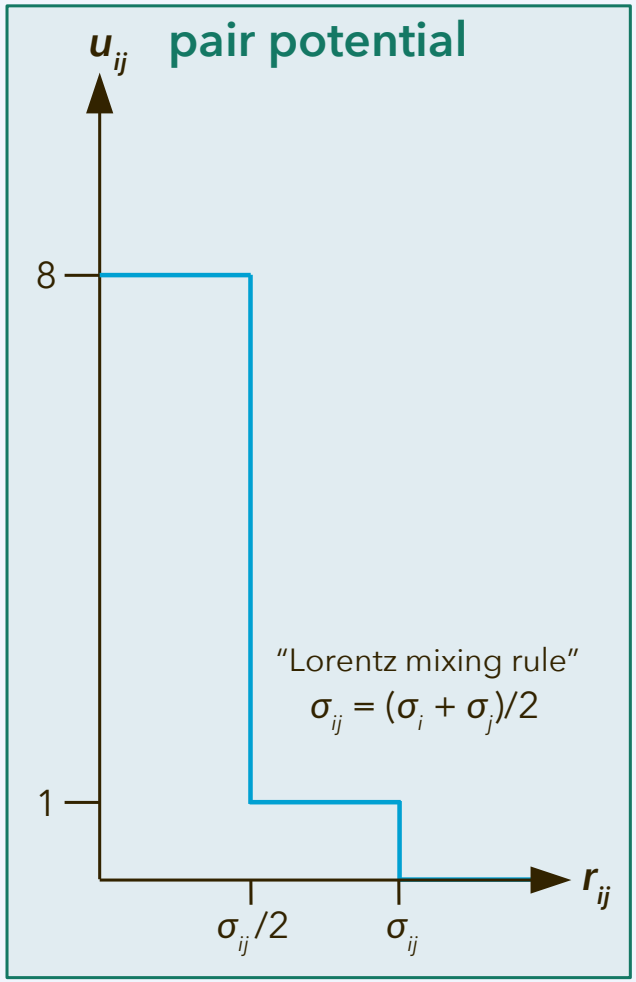
```
particle number N:  256
packing fraction:  0.777778
output file name:  benchmark-configuration.dat
max. size ratio:  3.33333
generating   5 spheres with diameter 3.33333 (occupied volume:   96.9627)
generating   2 spheres with diameter 2.33333 (occupied volume:   13.3033)
generating 122 spheres with diameter 1.33333 (occupied volume: 151.417  )
generating 127 spheres with diameter 1       (occupied volume:   66.497  )

occupied volume:  328.18
volume of the box:  421.946
size of the box:  7.50042
```

Proposal that avoids boring solutions from "stacking" of spheres (placing ≥2 spheres at the same coordinates):

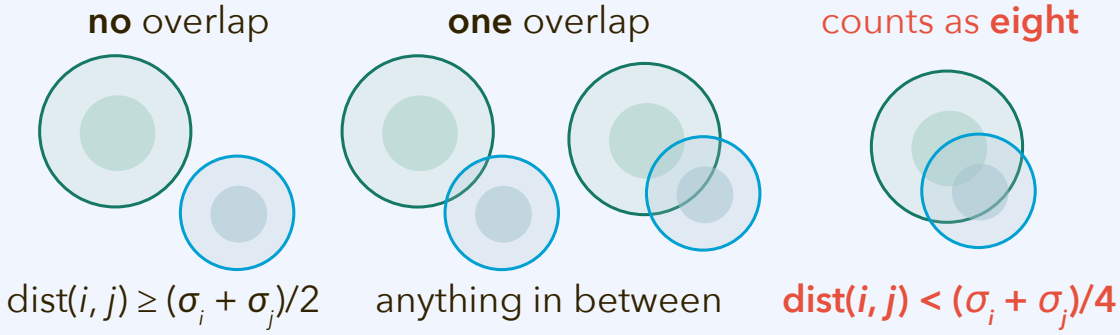


See implementation in **sphere-benchmark**, sphere.cpp, line 51.

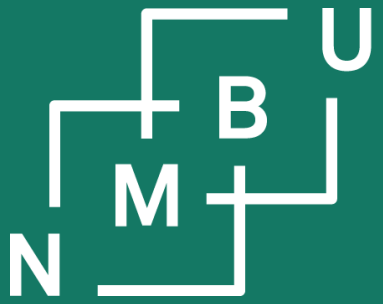
Benchmark scenario: Sphere overlap



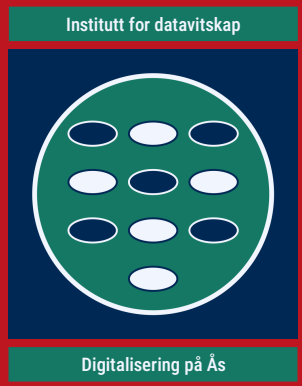

```
int Sphere::check_overlap(const Sphere* other, const double box_size[3]) const
{
  // square distance between the centre of i and the centre of j
  double square_distance = 0.0;
  for(int d = 0; d < 3; d++) {
    double dist_d = other->coords[d] - this->coords[d];

    // apply minimum image convention
    if(dist_d > 0.5*box_size[d]) dist_d -= box_size[d];
    else if(dist_d < -0.5*box_size[d]) dist_d += box_size[d];

    square_distance += dist_d*dist_d;
  }

  // is the square distance smaller than the square of the sum of radii?
  double sum_of_radii = 0.5 * (this->size + other->size);
  int overlap = 0;
  if(square_distance < 0.25*sum_of_radii*sum_of_radii) overlap = 8;  // soft shielding
  else if(square_distance < sum_of_radii*sum_of_radii) overlap = 1;  // normal overlap
  return overlap;
}
```



See implementation in **sphere-benchmark**, sphere.cpp, line 51.



Noregs miljø- og biovitskaplege universitet



4 Production

4.5 "#define" macros

4.6 Shared libraries

4.7 Benchmarks

4.8 CMake

Generating makefiles using CMake

CMake is used by many complex C/C++ projects that require developers or users to compile code on their systems, which may be very diverse. Typically:

cmake . && **make** && sudo make install

There, **CMake** generates the Makefile that is then used by **GNU make**.
We have done this before when we looked into the C++ interface to ROS.

Instructions for CMake are communicated through a file called **CMakeLists.txt**.

- CMake documentation: https://cmake.org/cmake/help/latest/
- CMake tutorial: https://cmake.org/cmake/help/latest/guide/tutorial/

CMake can be helpful if your project has a complex system of dependencies, or compile-time case distinctions are needed beyond what you can implement in a simple way using GNU make; *e.g.*, embedded system cross-compiling.

CMakeLists.txt commands

Create a simple CMake project (**cmake-dirgraph**):

- Copy the "**directed-graph**" code into the subdirectory **./src** of some working folder.
- Remove the original Makefile.
- Write **CMakeLists.txt** (see commands[1]) in the main folder and in the ./src folder.
- Calling "**cmake .**" in the main working folder generates Makefiles in both folders.

```
cmake_minimum_required(
  VERSION 3.14
)
project(
  dirgraph
    VERSION 1.0.0
    LANGUAGES CXX
)
set(AUTHOR "Martin Horsch")
set(CMAKE_CXX_STANDARD 17)
add_subdirectory(src)
```

./CMakeLists.txt

```
set(EXECUTABLE_OUTPUT_PATH ../bin)
add_executable(dirgraph graph.cpp query.cpp run-graph.cpp)
```

src/CMakeLists.txt

Now **make** will automatically call g++ with the right options and flags.

[1]See: https://cmake.org/cmake/help/latest/manual/cmake-commands.7.html

CMake support for unit tests

Unit tests are generally a helpful debugging tool in complex development projects. Here they can also help the user verify that everything worked well.

CMakeLists.txt (**cmake-dirgraph**)

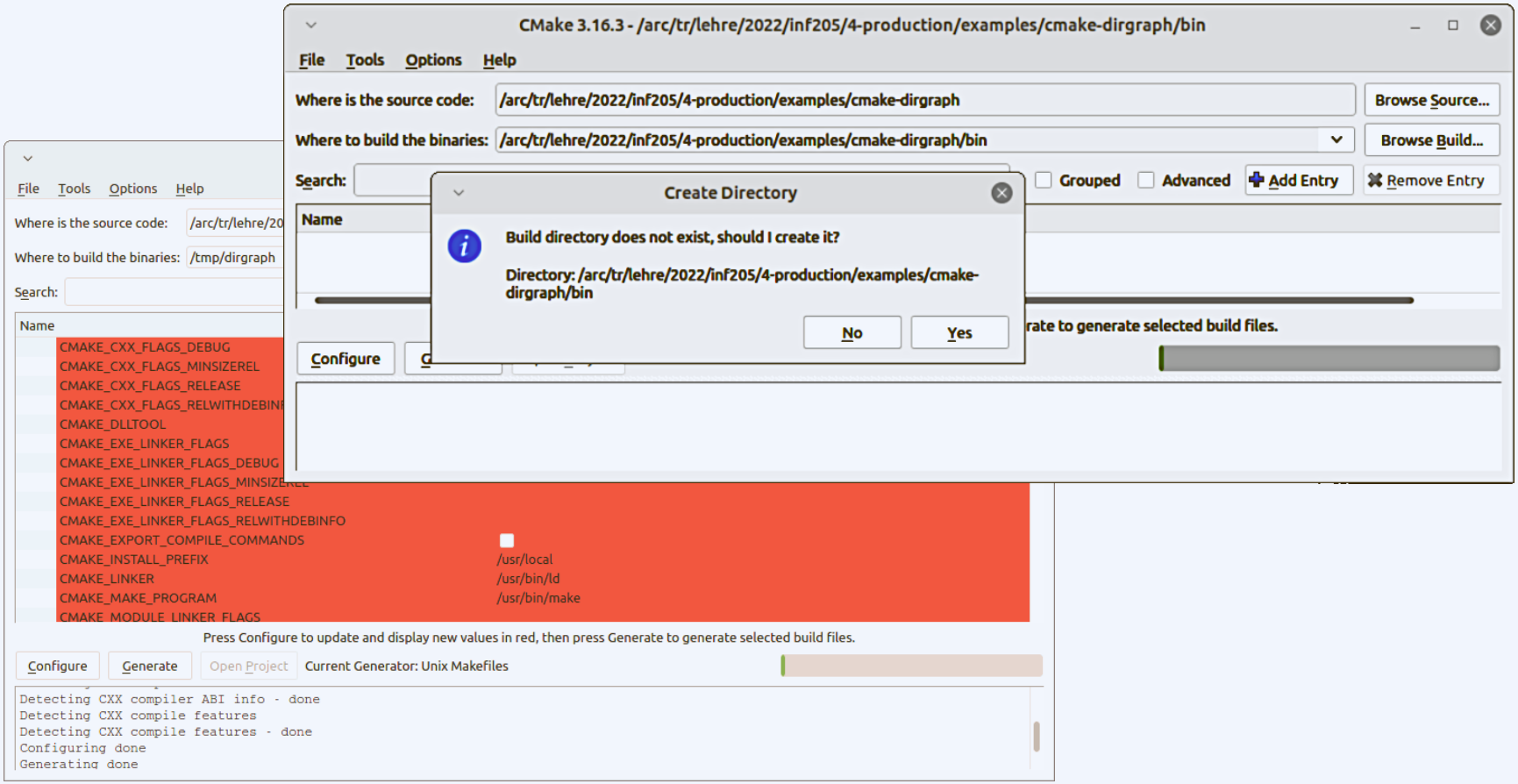
```
enable_testing()
add_test(
  NAME example_graph
  COMMAND dirgraph kb.dat query.dat
  WORKING_DIRECTORY data
)
set_tests_properties(
  example_graph
  PROPERTIES
  PASS_REGULAR_EXPRESSION "<INF200 2022H>[ \t\r\n]*<Rune Grønnevik>"
  PASS_REGULAR_EXPRESSION "<INF205 2023H>[ \t\r\n]*<Trine Næss Henriksen>"
  PASS_REGULAR_EXPRESSION "<KJM230 2023V>[ \t\r\n]*<Heidi Rudi>"
)
```

^	Matches at beginning of input
.	Matches any single character
[Xy2]	Any of the characters X, y, or 2
[^vV]	Any character other than v or V
[C-F]	Any of the characters C, D, E, or F
*****	Preceding pattern occurs >= 0 times
+	Preceding pattern occurs >= 1 time
?	Optional (occurs 0 or 1 times)
|	Disjunction ("or")

https://cmake.org/cmake/help/latest/manual/cmake-properties.7.html#test-properties

CMake GUI

Graphical interface to CMake: **cmake-gui**



Noregs miljø- og biovitskaplege universitet

Institutt for datavitskap

Digitalisering på Ås

Conclusion

INF205

16th November 2022

Update: Presentation schedule

Norwegian University of Life Sciences

Thursday, 1st December 2022

Black-white images and binary classification problems

10.00 – 10.20: Group #9
10.30 – 10.50: Group #23
11.00 – 11.20: Group #26

Graph querying: Paths and cycles in graphs

12.00 – 12.20: Group #8
12.30 – 12.50: Group #10
13.00 – 13.20: Group #17
13.30 – 13.50: Group #20

Wednesday, 7th December 2022

Spherical particle configurations

14.15 – 14.35: Group #1
14.40 – 15.00: Group #6
15.15 – 15.35: Group #12
15.40 – 16.00: Group #19

Friday, 9th December 2022

Various topics

12.30 – 12.50: Group #2
13.00 – 13.20: Group #3

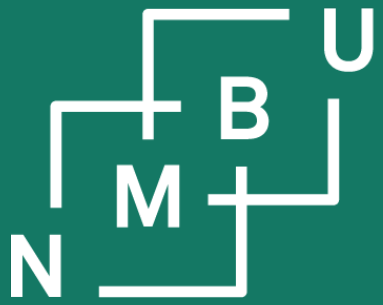
Thursday, 15th December 2022

Various topics

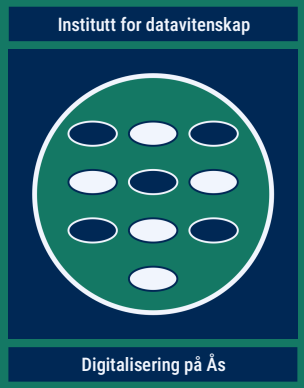
10.30 – 10.50: Group #4
11.00 – 11.20: Group #5
11.30 – 11.50: Group #7

Black-white images

12.30 – 12.50: Group #13
13.00 – 13.20: Group #16
13.30 – 13.50: Group #25



Norges miljø- og biovitenskapelige universitet



INF205
Resource-efficient programming

4 Production

4.5 "#define" macros
4.6 Shared libraries
4.7 Benchmarks
4.8 CMake