

IN3200/IN4200: High-Performance Computing & Numerical Projects

*Course overview & quick recap of serial
programming*

Xing Cai

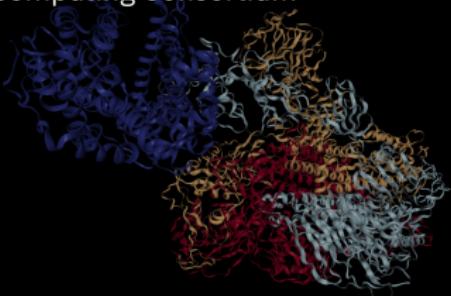
Simula Research Laboratory & University of Oslo, Norway

Spring 2021

Motivation from the real life

COVID-19 HPC Consortium Who We Are Collaborations Projects News & Press Blog Apply

The COVID-19 High Performance Computing Consortium



Bringing together the Federal government, industry, and academic leaders to provide access to the world's most powerful high-performance computing resources in support of COVID-19 research.

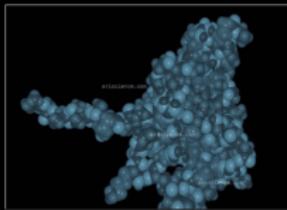
95 Projects 6.8m CPU cores

Active

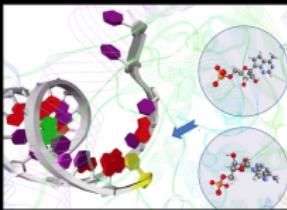
Fighting COVID-19 will require extensive research in areas like bioinformatics, epidemiology, and molecular modeling to understand the threat we're facing and to develop strategies to address it.

Here are some of our projects.

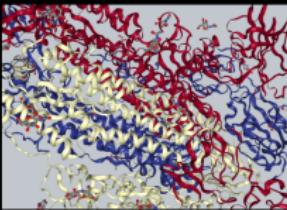
[See all](#)



SARSCOV2/COVID19 protein interruption search with existing active



The Competition of Antiviral Drugs with ATP to Inhibit the SARS-CoV-2 RNA-



High-Performance Causal Inference for COVID-19 Mitigation and Response

Real-life example 1

PHYS.ORG

≡
Topics

Week's top Latest news

Nanotechnology Physics Earth Astronomy & Space Technology

[Home](#) / [Chemistry](#) / [Biochemistry](#)
[Home](#) / [Chemistry](#) / [Analytical Chemistry](#)



11



49



Share

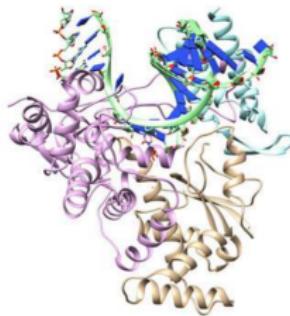


Email

SEPTEMBER 19, 2020

Pulling the plug on the coronavirus copy machine

by Jorge Salazar, Texas Advanced Computing Center



<https://phys.org/news/2020-09-coronavirus-machine.html>

Real-life example 1 (cont'd)

- To model key proteins used by coronavirus for its reproduction
- Use basic math and physics of Newton's equations and quantum mechanics to calculate the properties of these proteins
- HPC & supercomputers allow much faster simulations
- Goal: finding ways to improve COVID-19 drugs such as remdesivir
- Research team from University of North Texas



Simulations done on Frontera: No. 9 supercomputer in the world (according to TOP500 ranking in Nov. 2020)

<https://phys.org/news/2020-09-coronavirus-machine.html>

Real-life example 2



FEATURES

Fighting COVID-19 With the Power of Genomics and HPC

By Janet Morss | June 17, 2020

Researchers at Cardiff University are using the power of genomic sequencing and high performance computing to unlock the secrets of COVID-19.

In scientific laboratories around the world, efforts are under way to put the power of genomic sequencing and [high performance computing](#) (HPC) to work in the fight against COVID-19. At Cardiff University in Wales, a team of scientists is working with the COVID-19 Genomics UK Consortium (COG-UK), to unlock the secrets of the coronavirus that causes COVID-19.



YOU MAY ALSO LIKE

FEATURES

<https://www.delltechnologies.com/en-us/blog/fighting-covid-19-with-the-power-of-genomics-and-hpc/>

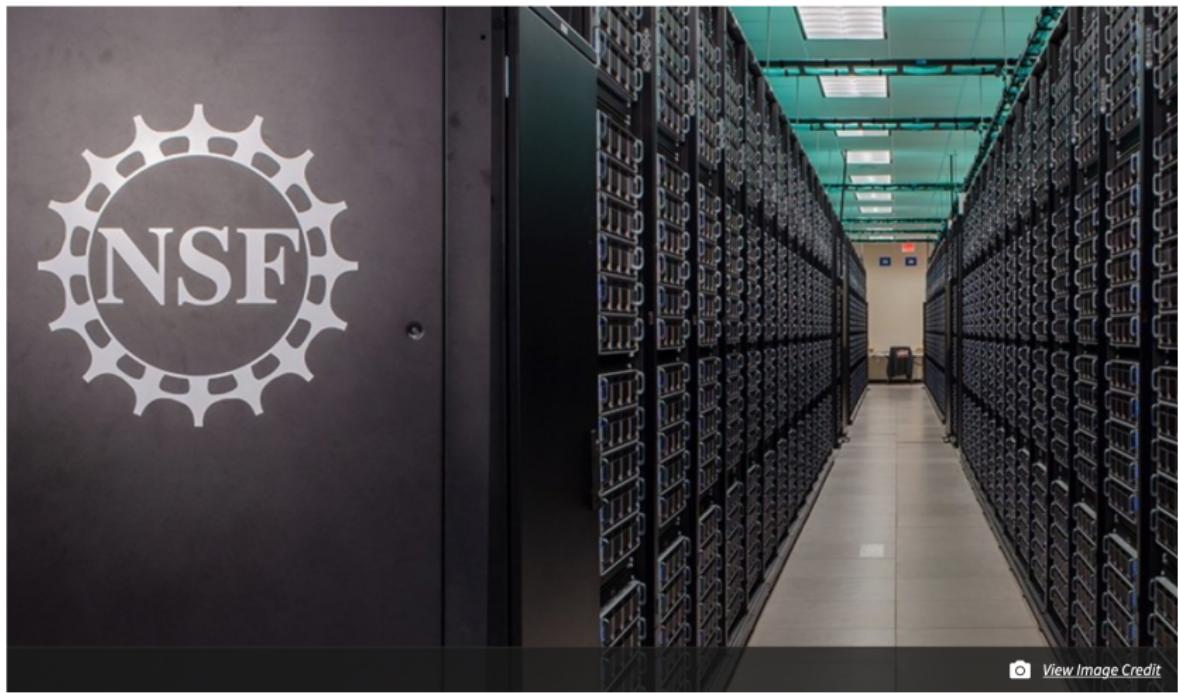
Real-life example 2 (cont'd)

- Large-scale, rapid genomic sequencing and analysis of the coronavirus
- Relying on a large shared system that provides 2.5 petabytes of HPC data storage, also a huge amount of memory (78 terabytes)
- Goal: Unlocking the secrets of the coronavirus
- Research team from Cardiff University in Wales



<https://www.delltechnologies.com/en-us/blog/fighting-covid-19-with-the-power-of-genomics-and-hpc/>

More about fighting COVID19 with HPC



 View Image Credit

Why are supercomputers so important for COVID-19 research?

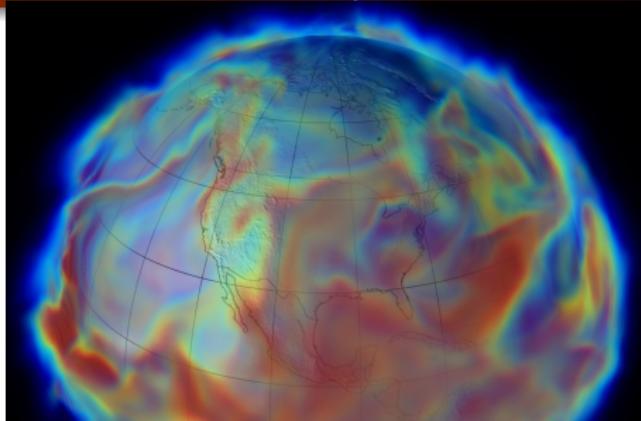
Spread the Word



General motivations for HPC

- Many problems in natural sciences can benefit from large-scale or huge-scale computations
 - more details
 - better accuracy
 - more advanced models
- The need for computing is ever-increasing
- However, standard laptop PCs or desktop computers are not powerful enough!

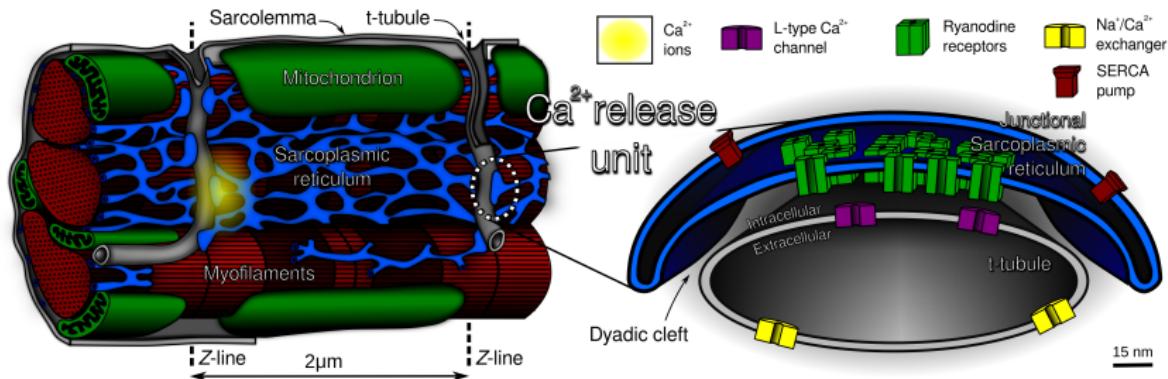
Huge computation example 1 (Climate Simulation)



NASA Center for Climate Simulation

- Earth surface area: $510,072,000 \text{ km}^2$
- If a spatial resolution of $1 \times 1\text{km}^2$ is adopted $\rightarrow 5.1 \times 10^8$ (510 million) small patches
- If a spatial resolution $100 \times 100\text{m}^2$ is adopted $\rightarrow 5.1 \times 10^{10}$ (51 billion) small patches
- Additional layers in the vertical direction
- High resolution in the time direction

Example 2 (Subcellular Calcium Dynamics Simulation)



- Size of one cardiac muscle cell: $100\mu\text{m} \times 10\mu\text{m} \times 10\mu\text{m}$
- Width of calcium release channels: 1 nanometer (nm)
- Ideal computational mesh resolution: 1 nm
- Computational mesh required: $10^5 \times 10^4 \times 10^4$ (in total 10^{13} computational voxels)
- Number of simulation time steps needed: $\sim 10^6$

An intereting video on YouTube



Supercomputing and eScience (Eng)

<https://www.youtube.com/watch?v=S9YPcPtPsuY>

Motivations (cont'd)

- Parallel computers are now everywhere!
 - CPUs nowadays have multiple “cores” on a chip
 - One computer may have several multicore chips
 - There are also accelerator-based parallel architectures — GPGPU
 - Clusters of different kinds



What do we learn in IN3200/IN4200?

High-performance computing (HPC) – an introduction

- Proper implementation of numerical algorithms
- Effective use of the hardware for numerical computations

After finishing the course, you should

- be able to write simple parallel programs with sufficiently good performance
- be able to learn more about advanced parallel programming on your own later

Part 1 of the course: Serial programming

- A brief architectural overview of modern cache-based microprocessors
- Inherent performance limitations of microprocessors
- Basic C programming
- Optimization strategies of serial code

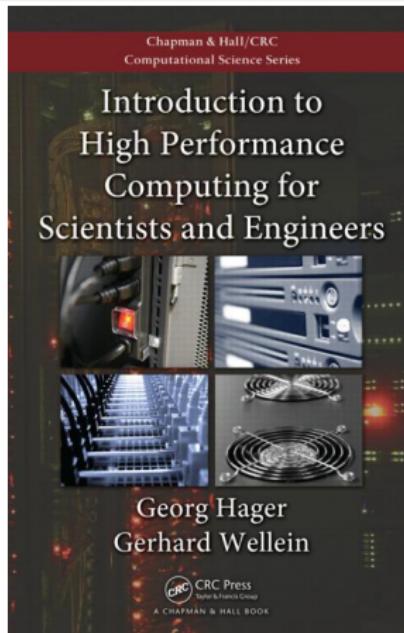
Part 2 of the course: Parallel programming

- Parallel computer architecture
- Theoretical considerations of parallel computing
- Shared-memory parallel programming (OpenMP)
- Distributed-memory parallel programming (MPI)

Why learning parallel programming?

- Parallel computing – a form of parallel processing by concurrently utilizing multiple computing units for one computational problem
 - shortening computing time
 - solving larger problems
- However ...
 - modern multicore-based computers are good at multi-tasking, but not good at automatically computing one problem in parallel
 - automatic parallelization compilers have had little success
 - special parallel programming languages have had little success
 - serial computer programs have to be modified or rewritten to utilize parallel computers
- Learning parallel programming is thus important!

Textbook



Georg Hager, Gerhard Wellein

**Introduction to High Performance Computing for Scientists
and Engineers**

1st Edition, CRC Press, ISBN 9781439811924

Teaching approaches

- Focus on fundamental issues
 - parallel programming = serial programming + finding parallelism + enforcing work division and collaboration
- Use of examples relevant for natural sciences
 - mathematical details are not required
 - understanding basic numerical algorithms is needed
 - implementing basic numerical algorithms is essential
- Hands-on programming exercises and tutoring
- English is the “official language” of the course, but students should feel free to ask questions, write emails/reports in Norwegian

Recapitulation of serial programming

What is serial programming?

- Roughly speaking, a computer program executes a sequence of operations applied to data structures
- A program is normally written in a programming language
- Data structures:
 - variables of primitive data types (`char`, `int`, `float`, `double` etc.)
 - variables of composite and abstract data types (`struct` in C, `class` in Java & Python)
 - array variables
- Operations:
 - statements and expressions
 - functions

Variables

- In a dynamically typed programming language (e.g. Python) variables can be used without declaration beforehand

```
a = 1.0
```

```
b = 2.5
```

```
c = a + b
```

- In statically typed languages (e.g. Java and C) declaration of variables must be done first

```
double a, b, c;
```

```
a = 1.0;
```

```
b = 2.5;
```

```
c = a + b;
```

Simple example

- Suppose we have temperature measurement for each hour during a day
- t_1 is the temperature at 1:00 o'clock, t_2 is the temperature at 2:00 o'clock, and so on.
- How to find the average temperature of the day?
- We need to first add up all the 24 temperature measurements:

$$T = t_1 + t_2 + \dots + t_{24} = \sum_{i=1}^{24} t_i$$

- The average temperature can then be calculated as $\frac{T}{24}$.

Simple example (cont'd)

- How to implement the calculations as a computer program?
- First, create an array of 24 floating-point numbers to store the 24 temperatures. That is, $t[0]$ stores t_1 , $t[1]$ stores t_2 and so on. Note that array index starts from 0!
- Sum up all the values in the array t

- Same syntax for the computational loop in Java & C:

```
T = 0;  
for (i=0; i<24; i++)  
    T = T + t[i];
```

- Syntax for Python:

```
T = 0  
for i in range(0,24):  
    T = T + t[i]
```

- Finally, $t_average = T/24.0;$

Similarities and differences between languages

- For scientific applications, arrays of numerical values are the most important basic building blocks of data structure
- Extensive use of `for`-loops for doing computations
- Different syntax details
 - allocation and deallocation of arrays
 - Java: `double[] v=new double[n];`
 - C: `double *v=malloc(n*sizeof(double));`
 - Python: `v=zeros(n,dtype=float64)` (using NumPy)
 - definition of composite and abstract data types
 - I/O

C as the main choice of programming language

- C is one of the dominant programming languages in computational sciences
- Syntax of C has inspired many newer languages (C++, Java, Python)
- Good computational efficiency
- C is ideal for using MPI and OpenMP (also GPU programming)
- We will thus choose C as the main programming language
- (Most of the textbook's coding examples are in Fortran, but many of the “performance-engineering” principles are the same.)

Some words about pointers in C

- A variable in a program has a name and type, its value is stored somewhere in the memory of a computer
- Type `*p` declares a pointer to a variable of datatype `Type`
- A pointer is actually a special type of variable, used to hold the memory address of a variable
- From a variable to its pointer: `int a; int *p; p = &a;`
- We can use a pointer to change the variable value `*p = 2;`
- A pointer can also be used to hold the memory address of the first entry of an array (such as returned by `malloc`)
- Array indexing: `p[0]`, `p[1]` ...
- Pointer arithmetic:

```
int *p = (int*)malloc(10*sizeof(int));  
int *p2 = p + 3; /* p2 is now pointing to p[3] */
```

Allocating multi-dimensional arrays

- Let's allocate a 2D array for representing a $m \times n$ matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

- Java:

```
double[][] A = new double[m][n];
```

- C:

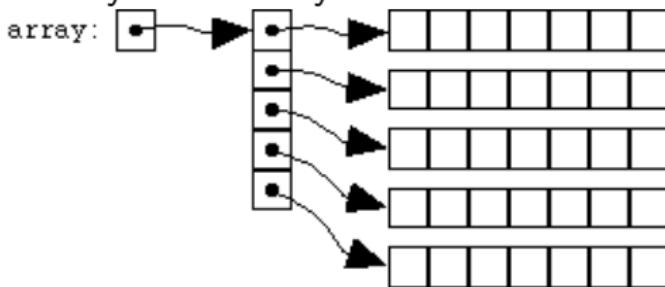
```
double **A = (double**)malloc(m*sizeof(double*));  
for (i=0; i<m; i++)  
    A[i] = (double*)malloc(n*sizeof(double));
```

- Same syntax in Java and C for indexing and traversing a 2D array

```
for (i=0; i<m; i++)  
    for (j=0; j<n; j++)  
        A[i][j] = i+j;
```

More about two-dimensional arrays in C (1)

- C doesn't have true multi-dimensional arrays, a 2D array is actually an array of 1D arrays



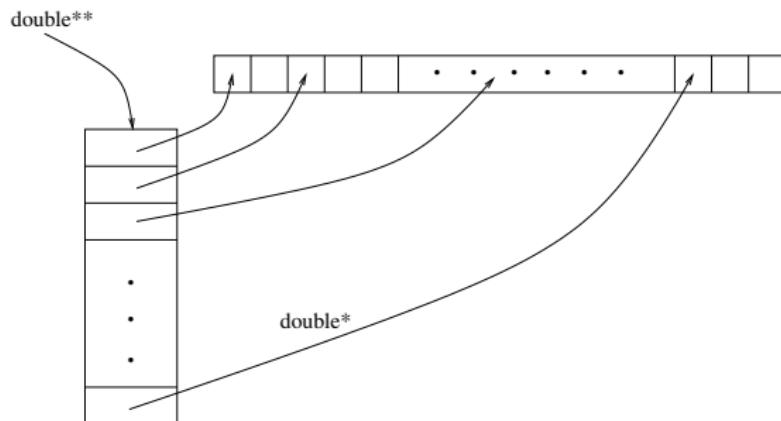
- $A[i]$ is a pointer to row number $i+1$
 - It is also possible to use static memory allocation of fix-sized 2D arrays, for example:
- ```
double A[10][8];
```

However, the size of the array is decided at compiler time (not runtime)

## More about two-dimensional arrays in C (2)

- Dynamic memory allocation of 2D arrays through e.g. malloc
- Another way of dynamic allocation, to ensure contiguous underlying data storage (for good use of cache):

```
double *A_storage=(double*)malloc(n*n*sizeof(double));
double **A = (double**)malloc(n*sizeof(double*));
for (i=0; i<n; i++)
 A[i] = &(A_storage[i*n]);
```



# Deallocation of arrays in C

- If an array is dynamically allocated, it is important to free the storage when the array is not used any more
- Example 1

```
int *p = (int*)malloc(n*sizeof(int));
/* ... */
free(p);
```

- Example 2

```
double **A = (double**)malloc(m*sizeof(double*));
for (i=0; i<m; i++)
 A[i] = (double*)malloc(n*sizeof(double));
/* ... */
for (i=0; i<m; i++)
 free(A[i]);
free(A);
```

- Be careful! Memory allocation and deallocation can easily lead to errors

## The form of a C program

- A program in C is made up of functions
- A stand-alone C program must at least implement function `main`, which will be executed by the operating system
- Functions are made up of statements and declarations
- Variables must be declared before usage
- Possible to use functions and variables declared in libraries

## Some syntax details in C

- Semicolon (;) terminates a statement
- Braces ({} ) are used to group statements into a block
- Square brackets ([]) are used in connection with arrays
- Comments can be added between /\* and \*/

# Functions in C

- Function declaration specifies name, type of return value, and (optionally) a list of parameters
- Function definition consists of declaration and a block of code, which encapsulates some operation and/or computation

```
return_type function_name (parameter declarations)
{
 declarations of local variables
 statements
}
```

## Function arguments

- All arguments to a C function are passed by value
- That is, a copy of each argument is passed to the function

```
void function test (int i) {
 i = 10;
}
```

The change of *i* inside *test* has no effect when the function returns

- Passing pointers as function arguments can be used to get output

```
void function test (int *i) {
 *i = 10;
}
```

The change of *i* inside *test* now has effect

## Function example 1: swapping two values

```
void swap (int *a, int *b)
{
 int tmp;
 tmp = *a;
 *a = *b;
 *b = tmp;
}
```

## Function example 2: smoothing a vector

- We want to smooth the values of a vector  $v$  by the following formula:

$$v_i^{\text{new}} = v_i + c(v_{i-1} - 2v_i + v_{i+1}), \quad 1 \leq i < n - 1$$

where  $c$  is a constant

```
void smooth (double *v_new, double *v, int n, double c)
{
 int i;
 for (i=1; i<n-1; i++)
 v_new[i] = v[i] + c*(v[i-1]-2*v[i]+v[i+1]);
 v_new[0] = v[0];
 v_new[n-1] = v[n-1];
}
```

- Similar computations occur frequently in numerical computations

## Function example 3: matrix-vector multiplication

- We want to compute  $\mathbf{y} = \mathbf{Ax}$ , where  $\mathbf{A}$  is a  $m \times n$  matrix,  $\mathbf{y}$  is a vector of length  $m$  and  $\mathbf{x}$  is a vector of length  $n$ :

$$y_i = A_{i1}x_1 + A_{i2}x_2 + \dots + A_{in}x_n = \sum_{j=1}^n A_{ij}x_j, \quad 1 \leq i \leq m$$

```
void mat_vec_prod (double **A, double *y, double *x,
 int m, int n)
{
 int i,j;
 for (i=0; i<m; i++) {
 y[i] = 0.0;
 for (j=0; j<n; j++)
 y[i] += A[i][j]*x[j];
 }
}
```

## Example of a complete C program

```
#include <stdio.h> /* import standard I/O functions */\n\nint myfunction(int x) /* define a function */\n{\n int r;\n r = x*x + 2*x + 3;\n return r;\n}\n\nint main (int nargs, char** args)\n{\n int x,y;\n x = atoi(args[1]); /* read x from command line */\n y = myfunction(x); /* invoke myfunction */\n printf("x=%d, y=%d\n",x,y);\n return 0;\n}
```

# Compilation

- Suppose a file named `first.c` contains the C program
- Suppose we use GNU C compiler `gcc`
- Step 1: Creation of a file of object code:

```
gcc -c first.c
```

An object file named `first.o` will be produced.

- Step 2: Creation of the executable:

```
gcc -o run first.o
```

The executable will have name `run`.

- Alternatively (two steps in one),

```
gcc -o run first.c
```

- Better to use the 2-step approach for complex examples

## Some important compiler options

- During compilation:
  - Option -O turns on optimization flag of the compiler
  - Option -c produces an object file for each source file listed
  - Option -Ixxx suggests directory xxx for search of header files
- During linkage:
  - Option -lxxx links with a specified library with name libxxx.a or libxxx.so
  - Option -Lxxx suggests directory xxx for search of library files
  - Option -o specifies the name of the resulting executable