High Performance Parallel Computing 2023

Introduction to C++ and Tools

Markus Jochum & Troels Haugbølle (slides courtesy Rasmus Munk)

Niels Bohr Institute
University of Copenhagen

Copenhagen, February 6, 2023

Practical Information

- Teachers

  Troels Haugbølle (`haugbel@nbi.ku.dk`)
  Markus Jochum (`mjochum@nbi.ku.dk`)
  Jorge Expósito Patiño (`jepa@di.ku.dk`)
  Roman Nuterman (`nuterman@nbi.ku.dk`)

- Location

  see Absalon - Course Overview

- Lectures

  Monday 13:15 - 16:00
  Wednesday 10:15 - 12:00

- Exercises

  Wednesday 13:15 - 16:00

Modules

Week 1: Basic architecture and programming (C++11)

Week 2: Vectorization – architecture and programming (OpenMP SIMD)

Week 3: Task Farming (MPI Master-Worker)

Week 4: Shared memory architectures and programming (OpenMP)

Week 5: GPUs and many-Core architecture (OpenACC)

Week 6: Distributed Memory and Networked Architectures (MPI)

Week 7: Project Week

Examination

- Requirement
  - 6 assignments, reports in groups of up to 3 persons (max three pages) 6 x 10%
  - 1 week-long project in groups of 4 to 6 persons (max 10 pages) 40%
- Assignment 1: Epidemic Model SIR (C++ programming)

  Deadline 12/2
- Assignment 2: Molecular Dynamics (Vectorization and Memory layout)

  Deadline 19/2
- Assignment 3: Particle Physics Electron data (Task Farming with MPI Master-Worker)

  Deadline 26/2
- Assignment 4: Seismology (Shared Memory parallelisation with OpenMP)

  Deadline 5/3
- Assignment 5: Shallow Water (GPU acceleration with OpenACC)

  Deadline 12/3
- Assignment 6: Climate Model (Distributed Memory with MPI)

  Deadline 19/3

Today's Lecture

- Introduction of C++11
- Makefile
- Debugging
- Jupyter – use the cloud

What is High Performance Parallel Computing?

- Solving tightly coupled problems using tightly coupled systems
- Handle domain decomposition
- Handle communication latency
- Handle large data sets

What is C++?

- A general-purpose object-oriented programming language.
- Invented by Bjarne Stroustrup (1979).
- Extension of the C language (but not a superset).
- Initially, the language was called “C with Classes”.
- Encapsulates both high- and low-level language features.

Why C++?

Pros:

- Link compatible with C and FORTRAN
- Performance similar to C and FORTRAN
- Great Standard Library
- Static typed
- Predictable performance e.g. no garbage collector
- High-level containers
- Generic programming through templates
- Object-oriented

Cons:

- Object-oriented
- HUGE and Complex!

A Very Short Introduction to C++

- We will only cover a fraction of C++
- `en.cppreference.com`
- `www.cplusplus.com`
- `erlerobotics.gitbooks.io/erle-robotics-cpp-gitbook`
- `www3.ntu.edu.sg/home/ehchua/programming/index.html`

Python versus C++

```python
def abs_add(a, b):
  result = a + b
  if result < 0:
    result = -result

  return result
```

```cpp
int abs_add(int a, int b) {
  int result = a + b;
  if(result < 0) {
    result = -result;
  }
  return result;
}
```

- Newline versus semicolons
- Tabs versus brackets
- Dynamic typed versus static typed

Hello World Example

```
#include <iostream>

int main(int argc, char **argv) {
  std::cout << "Hello World!\n";
  return 0;
}
```

- The execution starts at `main()`
- Use `#include <...>` to include libraries
- Use `std::cout` to write to standard out
- Use `"\n"` to write newline
- `int argc, char **argv` are the command line arguments

```cpp
#include <cstdint>
#include <complex>

char;              // 1 byte
short;             // 2 bytes (compiler specific)
int; long int;     // 4 bytes (compiler specific)
long long int;     // 8 bytes (compiler specific)
float;             // 4 bytes
double;            // 8 bytes

// Unsigned versions
unsigned char; unsigned short; unsigned int...

// C99 types from <cstdint>
int8_t; int16_t; int32_t; int64_t;
uint8_t; uint16_t; uint32_t; uint64_t;

// Complex types from <complex>
std::complex<float>  //  8 bytes
std::complex<double> // 16 bytes
```

References

```
void add_inplace(int &a, int b) {
    a = a + b;
}
int main() {
    int v = 0;  // Declare a new integer named `v`
    add_inplace(v, 42); // Updating the value of `v`
    std::cout << "The new value of `v` is: " << v << "\n";
}
```

- `int v = 0;` both declare and initiate a new integer.
- `add_inplace` doesn't return anything, instead it updates `value` in-place.
- The type `int &` is a *reference* to an integer.

Constant Reference

```cpp
void add_inplace(int &a, const int &b) {
    a = a + b;
}
int main() {
    int v = 0;  // Declare a new integer named `v`
    add_inplace(v, 42); // Updating the value of `v`
    std::cout << "The new value of `v` is: " << v << "\n";
}
```

The type `const int &` is a *constant* reference to an integer.

```cpp
int main() {
    int v = 0;
    int &ref = v;
    const int &cref = v;
}
```

Pointers

```
void add_inplace(int *a, const int b) {
    *a = *a + b;
}
int main() {
    int v = 0;
    add_inplace(&v, 42); // Updating the value of `v`
    std::cout << "The new value of `v` is: " << v << "\n";
}
```

- The type `int *` is a *pointer* to an integer.
- The `&v` operator provides the *pointer* to ‘v‘

For Loop

```cpp
int s = 0;
for(int i=0; i<10; ++i) {
    s += i;
}
```

```python
s = 0
for i in range(10):
    s += i
```

While Loop

```cpp
int s = 0;
int i = 0;
while(i < 10) {
    s += i;
    ++i;
}
```

```python
s = 0
i = 0
while i < 10:
    s += i
    i += 1
```

```cpp
#include <vector>  // Standard Vector Class

// Empty vector of integers: []
std::vector<int> vec;

// Uninitiated vector of integers of size three
std::vector<int> vec(3);
std::vector<int> vec{3};

// Initiated vector of integers of size three: [42, 42, 42]
std::vector<int> vec(3, 42);

// Initiated vector of integers of size three: [2, 4, 3]
std::vector<int> vec = {2, 4, 3};

// Return the size of the vector
vec.size()

// Append a value to the vector
vec.push_back(42)
```

For-each-loop

For each element `v` in `vec`, run the loop body.

```cpp
void f(std::vector vec) {
    int s = 0;
    for(int &v: vec) {
        s += v;
    }
}
```

```python
def f(vec):
    s = 0
    for v in vec:
        s += v
```

Question

```
int vec_sum(std::vector vec) {
    int s = 0;
    for(int &v: vec) {
        s += v;
    }
    return s;
}
```

How can we optimize the performance of `vec_sum`?

Answer - use a reference

```
int vec_sum(std::vector &vec) {
    int s = 0;
    for(int &v: vec) {
        s += v;
    }
    return s;
}
```

We can avoid copying `vec` by using a reference `std::vector &`.

Function Overload

```
int add_one(int a) {
    return a + 1;
}
float add_one(float a) {
    return a + 1;
}

// Print `43`
std::cout << add_one(42) << "\n";
// Print `5.2`
std::cout << add_one(4.2) << "\n";
```

Generic Programing – Template

```cpp
template<typename T>
T add_one(T a) {
    return a + 1;
}
// Print `43`
std::cout << add_one(42) << "\n";
// Print `5.2`
std::cout << add_one(4.2) << "\n";
// Print `5`
std::cout << add_one<int>(4.2) << "\n";
```

`std::vector<int>` is a templated class.

Class

```cpp
class Body {
public:
    double mass;
    double pos_x;
    double pos_y;
    double vel_x;
    double vel_y;
};
// Uninitiated instantiation of `Body`
Body my_body;
// Initiated instantiation of `Body`
Body my_body{42, 2.4, 1.4, 0, 0};
// Write to the `pos_x` attribute of `my_body`
my_body.pos_x = 4.2;
```

Class Constructor

```
class Account {
private:
    double balance;
public:
    Account(double start_balance) : balance{start_balance} {
        std::cout << "Start balance is: " << balance << "\n";
    }
};
// Create a new account with a balance of 4.2
Account my_account{4.2};
my_account.balance; // ERROR!
```

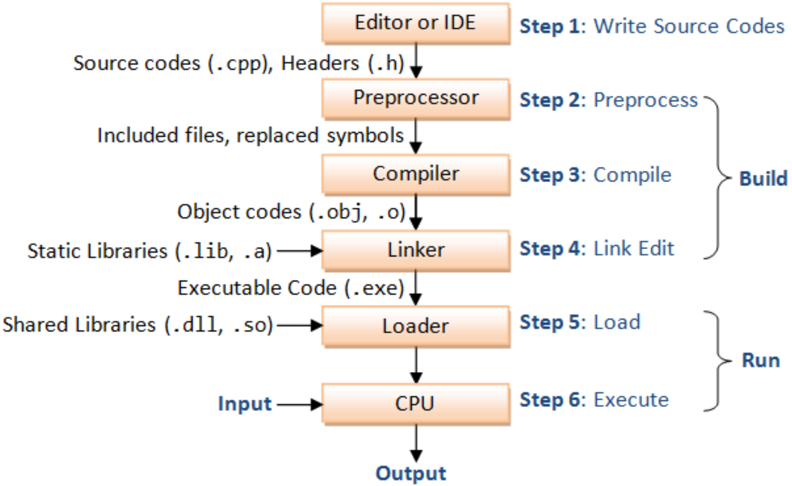
Class Methods

```cpp
class Account {
private:
    double balance;
public:
    Account(double start_balance) : balance{start_balance} {
        std::cout << "Start balance is: " << balance << "\n";
    }

    void deposit(double amount) {
        balance += amount;
    }

    void withdraw(double amount) {
        balance -= amount;
    }
};
Account my_account{4.2};
my_account.withdraw(2);
```

The Compiling Process



Compilation

```cpp
#include <iostream>

int main(int argc, char **argv) {
  std::cout << "Hello World!\n";
  return 0;
}
```

```
g++ -std=c++11 -O3 -g hello.cpp -o hello
```

- GNU Compiler Collection `gcc`. Alternatives: `clang`, `icc`, `pgcc`
- Optimization flag `-O0`, `-O1`, `-O2`, `-O3`
- Debug symbols `-g`

```makefile
LIB ?= -lm
INC ?= -I/opt/hpc_course/include
FLAGS ?= -O3 -DNDEBUG -g -Wall

all: nbody_seq nbody_acc

nbody_seq: nbody_seq.cpp
	pgc++ $(FLAGS) $(LIB) $(INC) -o nbody_seq nbody_seq.cpp

nbody_acc: nbody_acc.cpp
	pgc++ -acc -ta=multicore \
		$(FLAGS) $(LIB) $(INC) -o nbody_acc nbody_acc.cpp

clean:
	rm -f nbody_seq nbody_acc *.o

.PHONY: clean all
```

```
make
```

Printing Debugging

Follow the state of you program by using `std::cout`.

```
#include <iostream>

int abs_add(int a, int b) {
    int result = a + b;
    if(result < 0) {
        std::cout << "result is negative: " << result << "\n";
        result = result;
    }
    std::cout << "result of abs_add(): " << result << "\n";
    return result;
}
```

Assertions

Insert `assert()`

```
#include <cassert>

int abs_add(int a, int b) {
  int result = a + b;
  if(result < 0) {
    result = result;
  }
  assert(result >= 0);
  return result;
}
```

NOT setting `-DNDEBUG` enables `assert()`. Use `-Wall` to get all compiler warnings.

```
gcc -DNDEBUG -Wall
```

The GNU Project Debugger (GDB)

```
gdb --args /my_program arg1 arg2 arg3
```

- `run` run your program.
- `step` run to the next operation.
- `next` run to the next line.
- `break` set breakpoint.
- `list` show current source code.
- `bt` show backtrace.
- `p` run command and show the result e.g. `p my_var` will show the value of `my_var`.

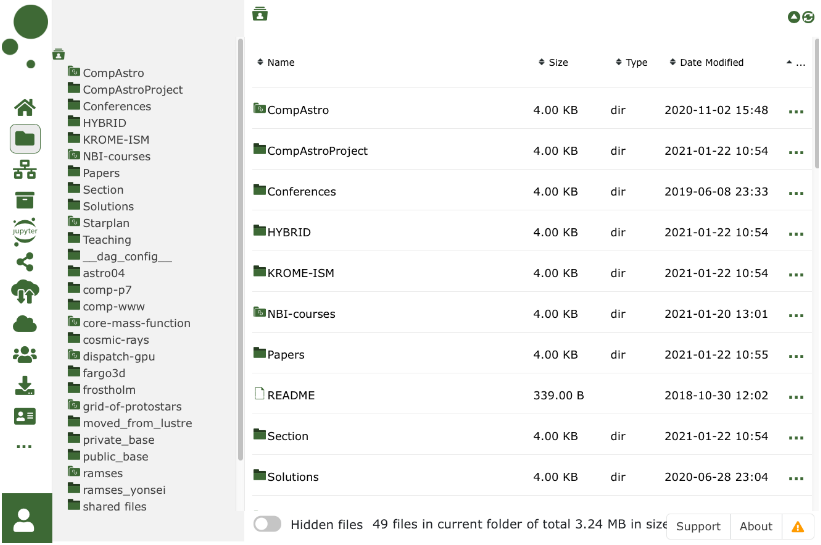
Valgrind

Check your program for illegal memory accesses and memory leaks

```
valgrind /my_program arg1 arg2 arg3
```

```
==21366== HEAP SUMMARY:
==21366==     in use at exit: 26,935 bytes in 31 blocks
==21366==   total heap usage: 535 allocs, 504 frees, 98,010 bytes allocated
==21366==
==21366== LEAK SUMMARY:
==21366==    definitely lost: 0 bytes in 0 blocks
==21366==    indirectly lost: 0 bytes in 0 blocks
==21366==      possibly lost: 0 bytes in 0 blocks
==21366==    still reachable: 26,935 bytes in 31 blocks
==21366==         suppressed: 0 bytes in 0 blocks
==21366== Rerun with --leak-check=full to see details of leaked memory
==21366==
==21366== For counts of detected and suppressed errors, rerun with: -v
==21366== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Jupyter


CompAstro
CompAstroProject
Conferences
HYBRID
KROME-ISM
NBI-courses
Papers
Section
Solutions
Starplan
Teaching
__dag_config__
astro04
comp-p7
comp-www
core-mass-function
cosmic-rays
dispatch-gpu
fargo3d
frostholm
grid-of-protostars
moved_from_lustre
private_base
public_base
ramses
ramses_yonsei
shared files
jupyter
Name
Size
Type
Date Modified
CompAstro 4.00 KB dir 2020-11-02 15:48
CompAstroProject 4.00 KB dir 2021-01-22 10:54
Conferences 4.00 KB dir 2019-06-08 23:33
HYBRID 4.00 KB dir 2021-01-22 10:54
KROME-ISM 4.00 KB dir 2021-01-22 10:54
NBI-courses 4.00 KB dir 2021-01-20 13:01
Papers 4.00 KB dir 2021-01-22 10:55
README 339.00 B 2018-10-30 12:02
Section 4.00 KB dir 2021-01-22 10:54
Solutions 4.00 KB dir 2020-06-28 23:04
Hidden files
49 files in current folder of total 3.24 MB in size
Support
About

Select a Jupyter Service

DAG MODI

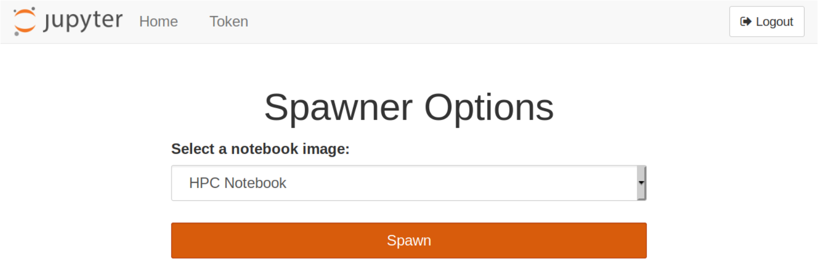
Service Description

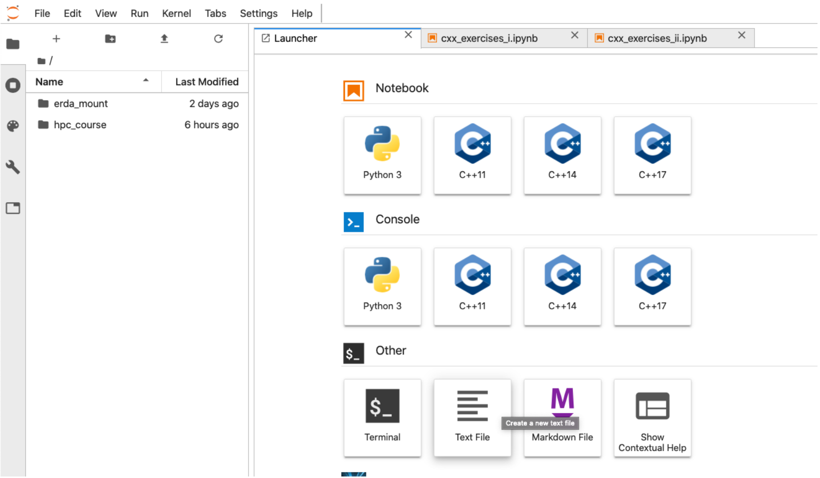
Data Analysis Gateway or DAG provides a set of interactive data analysis nodes for intermediate computation, which can be completed in a short timeframe.
This means that any spawned instance is limited to 2 hours of inactivity before it will be terminated. DAG instances have access to 8 compute cores and 8GB of memory

The spawned instances are non-persistent, meaning that any change made during a session is lost once the server is terminated.
The only exception to this is the data that is saved in the provided mount directory (i.e. ~/work)

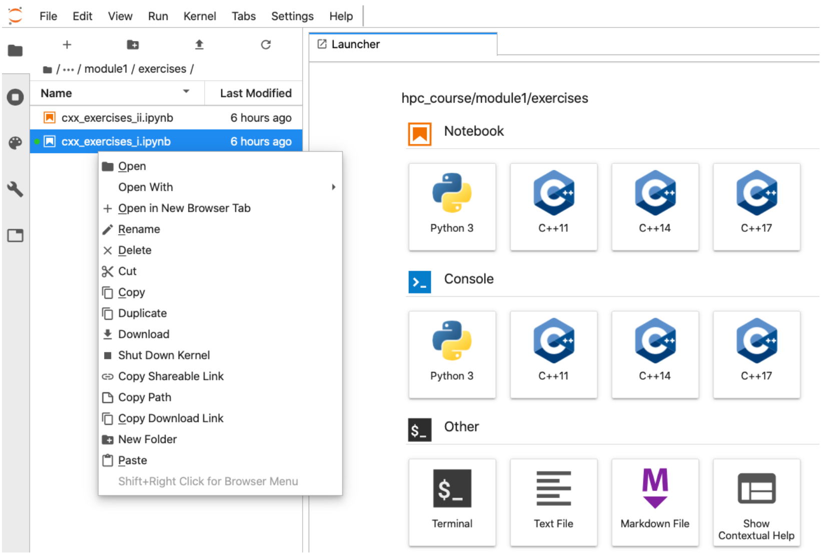
For more information about how you can ease the task of configuring your instances and our future roadmap for allowing customization, check out the FAQ section "How do I install and run software XYZ in Jupyter?" at ERDA FAQ

Start DAG


jupyter
Home
Token
Logout
Spawner Options
Select a notebook image:
HPC Notebook
Spawn



- **NB:** remember to move your files into `erda_mount`



- **NB:** move `cxx_exercises_X.ipynb` into your `erda_mount`!

File Edit View Run Kernel Tabs Settings Help

Launcher cxx_exercises_i.ipynb

Markdown

C++ Exercises I

NB: Remember to move this notebook into `/erda_mount` before starting.

1) Print hello world

Hint: remember to include `iostream` and end the output with the newline character: `\n`

[ ]:

2) Create a for-loop that print all even number from 0 to 20

[ ]:

3) Create a while-loop that print all odd number from 0 to 20

[ ]:

4) Write a function that takes one argument, the radius, and returns the volume of a sphere

[ ]:

5) Write a function that takes two arguments, the dividend and the divisor, and returns the quotient and remainder.

[ ]: