Concurrency vs. parallelism

- Concurrency: Two tasks are independent.
- Parallelism: Two tasks are executed simultaneously.

Tasks and threads

```cpp
using namespace std;
int main(){
  auto f1 = [](){ cout << "Task 1\n"; };
  auto f2 = [](){ cout << "Task 2\n"; };
  thread t1(f1);
  // This line does not
  // wait for f1 to finish
  thread t2(f2);
  t1.join();
  t2.join();
}
```

- Must join (blocking) before destruction.
- Asynchronous.

Waiting

```
using namespace std;
this_thread::sleep_for(chrono::seconds(1));
```

```
$ time ./a.out
```

Mutexes and locks

```
using namespace std;
int main(){
  mutex m;
  auto f1 = [](){
    scoped_lock l{m};
    cout << "Task 1\n";
  };
  auto f2 = [](){
    scoped_lock l{m};
    cout << "Task 2\n";
  };
  thread t1(f1);
  thread t2(f2);
  t1.join();
  t2.join();
}
```

- Only one thread can lock a mutex.

Joining thread

```
using namespace std;
int main(){
  auto f1 = [](){ /*...*/ };
  auto f2 = [](){ /*...*/ };
  jthread t1(f1);
  jthread t2(f2);
}
```

- A `thread` with RAII.

Sharing data

```
using namespace std;
int main(){
  int data = 0;
  auto f1 = [&](){data++;};
  auto f2 = [&](){data++;};
  thread t1(f1);
  thread t2(f2);
  t1.join();
  t2.join();
  cout<<data<<endl;
}
```

Sharing data

```
using namespace std;
int main(){
  int data = 0;
  auto f1 = [&](){data++;};
  auto f2 = [&](){data++;};
  thread t1(f1);
  thread t2(f2);
  t1.join();
  t2.join();
  cout<<data<<endl;
}
```

- Error: Race condition

Atomics

```
using namespace std;
int main(){
  atomic<int> data = 0;
  auto f1 = [&](){data++;};
  auto f2 = [&](){data++;};
  jthread t1(f1);
  jthread t2(f2);
}
```

- No concurrent access.

Future and async

```
int foo(int value){
  this_thread::sleep_for(1s);
  return 42*value;
}
int main(){
  future<int> f = async(foo, 1);
  future<int> f2 = async(foo, 2);
  cout << f.get()+f2.get() << endl;
  return 0;
}
```

Easy parallelism: STL

```cpp
#include <execution>
#include <algorithm>
#include <vector>
int main() {
  std::vector v = {3, 1, 4, 1, 5, 9, 2, 6};
  auto policy = execution::seq;
  std::sort(policy, v.begin(), v.end());
}
```

- Sequential

Easy parallelism: STL

```cpp
#include <execution>
#include <algorithm>
#include <vector>
int main() {
  std::vector v = {3, 1, 4, 1, 5, 9, 2, 6};
  auto policy = execution::par;
  std::sort(policy, v.begin(), v.end());
}
```

- Parallel

Easy parallelism: STL

```cpp
#include <execution>
#include <algorithm>
#include <vector>
int main() {
  std::vector v = {3, 1, 4, 1, 5, 9, 2, 6};
  auto policy = execution::par_unseq;
  std::sort(policy, v.begin(), v.end());
}
```

- Parallel unsequenced

Easy parallelism: tbb

```
#include <tbb/parallel_for.h>
int main(){
  tbb::parallel_for(0, 10, [](int i){
    do_something(i);
  });
}
```

- Install via conda.
- Link with -ltbb.

Easy parallelism: tbb

```cpp
#include <tbb/parallel_sort.h>
int main(){
  std::vector v = {3, 1, 4, 1, 5, 9, 2, 6};
  tbb::parallel_sort(v.begin(), v.end());
}
```

- Install via conda.
- Link with -ltbb.

Easy parallelism: thrust

```
#include <thrust/device_vector.h>
#include <thrust/sort.h>
int main(){
  thrust::device_vector<int> v = {3, 1, 4, 9};
  thrust::sort(v.begin(), v.end());
}
```