Timing

It is often useful to know how long a computation takes.

Programmers are willing to spend lots of time to writing ill-maintainable code because they think it will be faster.

First measure! I have been surprised many times.

Invest in quality of algorithms, and in modulariy, not in low level optimization.

If due to your improvement, you loose extensibility, you run the risk that your code will be rewritten later. (because it could not be extended.) It is better to inefficiently meet requirements, then to fail to meet requirements.

(I show you some code that measures speed.)

Threading

Threading means that a program separates into different subprograms that execute separately. into different subprograms that execute separately.

There are three reasons why you might want to this.

1. Avoidance of waiting times. Often, a program has to wait for something to happen in the outside world. (User typing input, a file to read from disk, data over the internet to arrive.)

   Using threading, one thread can be waiting, while the other threads are still doing something useful.

2. Modern processors have multiple cores. Since the last 5-10 years, the speed of a single core has not increased much.

   If you compute in a single core, you may miss 75% of the capacity of a modern CPU.

3. Maintaining responsiveness during heavy calculations.

Threading

An object of type `std::thread` may contain a running thread, or nothing.

```
std::thread t;    // Create thread with nothing in it.

std::thread t2{ f, a1, ..., an };
   // Create a thread that runs f(a1, ..., an).
```

The **thread** object should exist until f terminates by itself.

If the thread object goes out of scope, then f is interrupted in an unfriendly way by the operating system, and your complete program will be terminated.

Use **join** to make the main thread wait for its subprocesses.

```
struct counter
{
   std::string s;

   void operator( ) ( unsigned int k ) const
   {
      std::cout << "Starting Count:\n";
      for( unsigned int i = 0; i < k; ++ i )
      {
         std::cout << s << " has counted to "
                   << i << " " << "\n";
      }
      std::cout << "\n";
   }
};
```

Linking

If you use threads, then link

```
thread : thread.o
        $(CPP) $(Flags) thread.o -pthread -o thread
```

If you want to know how many threads your hardware supports, use `thread::hardware_concurrency( )`.

The different threads run completely independently. Nothing can be assumed about how they are timed.

```
// In one thread:

std::cout << "hello world\n";

// In another thread:

std::cout << "good evening\n";
```

```
std::thread t1{f, a1 };
std::thread t2{f, a2 };
std::thread t3{a, a3 };

t1. join( ); // Wait until t1 is done.
t2. join( ); // Wait until t2 is done.
t3. join( ); // Wait until t3 is done.
```

We see that **thread** does not follow usual C^{++} semantics, where destroying means quitely returning resources.

Safethread

If you do not like that, use

```
struct safethread
{
   std::thread t;
   safethread( std::thread&& t );
   ~safethread( )
   {
      if( t. joinable( )) t. join( );
   }
}
```

Not joinable threads are threads that **(1)** were default constructed, bf (2 have been moved away, **(3)** were already joined.

Terminated processes are still joinable.

Data Races

Shared data between different threads can cause lots of problems.

Two threads write into same variable. The one that is last 'wins'.

One thread writes into a variable, another one reads. We don't know if reading happens before or after writing.

Data Races (2)

If variable v is complicated, (e.g. `std::map< >`), data races may lead to complete chaos:

Writing may be complicated, it may cause rearranging the tree.

During rearrangement, a read may happen (or another write.)

The effect is unpredictable.

Mutex

A mutex is an object that guarantees mutual exclusion.

```
std::mutex m;
   //  Make sure that all threads get a reference to m.

// In a thread:

m. lock( );     // It is guaranteed that only one thread
                // can lock the mutex at the same time.

...    // Perform my delicate operations.

m. unlock( );
```

What happens if you forget to **unlock( )**, or the **unlock( )** is skipped (due to an exception)?

Sometimes, getting access to a **mutex** is not important enough to wait for:

```
   if( m. try_lock( ))
   {
      Now we have the lock.

      m. unlock( );
   }
   else
   {
      We didn't get a lock, do something else instead.
   }
};
```

Dangers of Locking

Locking too long destroys efficiency and responsiveness.

Forgetting too unlock (or passing unlock because an exception is thrown) probably causes deadlock.