Moteurs de jeux, Cours 3

Optimisation et parallélisation

Université Montpellier 2

Rémi Ronfard

remi.ronfard@inria.fr

https://team.inria.fr/imagine/remi-ronfard/

Le but de cette présentation est de fournir tous les moyens mis à votre disposition afin de produire un moteur de jeu le plus efficace possible.

Il s'agit d'un cours compliqué, avec de nombreuses nouvelles notions.

Des notions que vous ne verrez jamais dans d'autres cours ..

Comment optimiser votre mémoire pour accélérer vos calculs ?

Que faire calculer à votre moteur de jeux ?

Comment sauvegarder les données ?

Comment communiquer entre différents composants ?

Comment valider votre travail ?

- La gestion de la mémoire est un crucial pour obtenir un moteur de jeu efficace. Pour cela il faut contrôler différents points:
  - La quantité d'information produite
  - Le type de données utilisé
  - La localisation de la donnée.

Mémoire

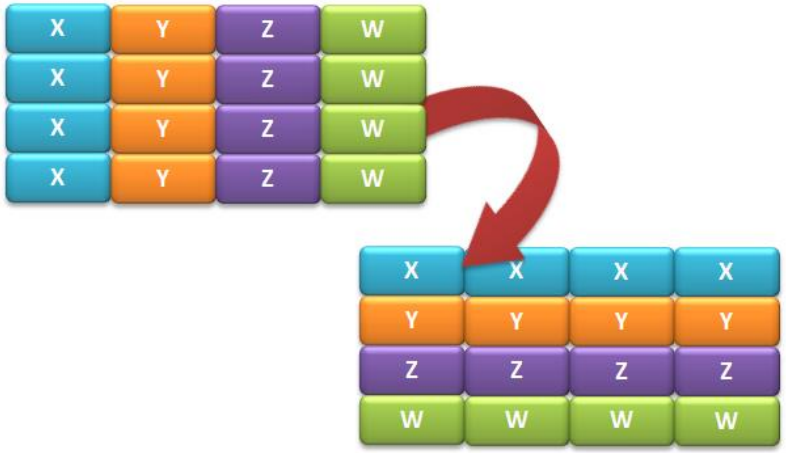
- Il faut mettre en place plusieurs stratégies:
  - Allouer des pools mémoires de tailles maitrisé
    - Eviter l'allocation multiple de petits éléments.
    - Favoriser la réduction du nombre d'allocation
  - Mettre en place une gestion efficace de la mémoire
    - Eviter la fragmentation de l'information
    - Evaluer les capacités de la mémoire de la machine cible
  - Surveiller les fuites mémoire

- Les opérations d'écriture et lectures en mémoire ont un coup.
  - Une évolution des architectures (32bits, 64 bits, …)
  - Evaluer la meilleur solution entre :
    - Entier
    - Float
    - Double
    - …
  - Ne pas surconsommer en mémoire
  - Réutiliser les bits non utile
  - Réutiliser la mémoire déjà alloué
  - Virgule fixe vs Virgule flottante
  - Précision des calculs (options compilateur, choix de la donnée)

- La localisation de la donnée est un point critique.
- Souvent une structure de données moins adapté à l'humain est plus efficace.
  - Privilège des structure de données SOA
    (Structure Of Array)
  - Eviter les données structurés en AOS
    (Array Of Structure)

```
struct {
   uint8_t r, g, b;
} AoS[N];

struct {
   uint8_t r[N];
   uint8_t g[N];
   uint8_t b[N];
} SoA;
```

- SOA mieux adapté pour utilisé la vectorisation
- Des compilateurs qui auto-vectorisent
- Un gain de temps de chargement dans le pipelines

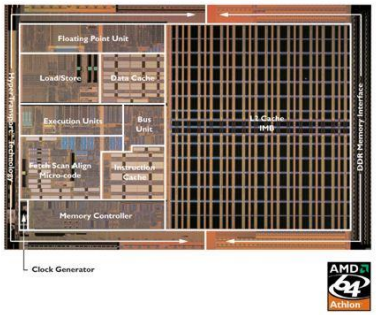


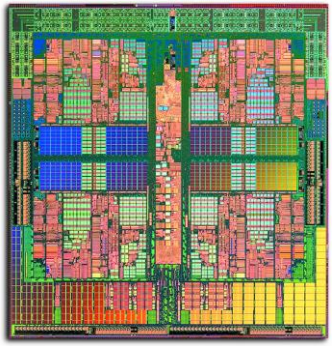
Calcul

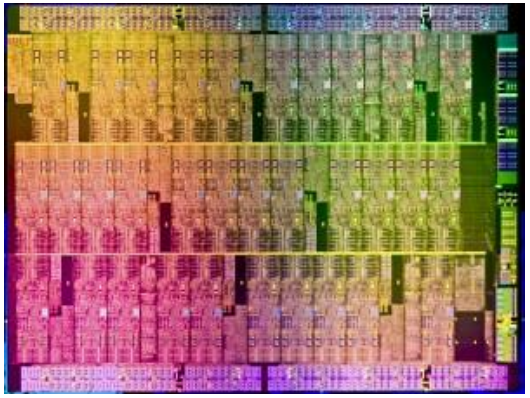
- Avant les machines étaient single-core.
- Maintenant, les machines sont multi-core.
- Demain, elles seront many-core.

- Loi de Moore:

“Number of transistors on integrated circuits doubles approximately every two years.“







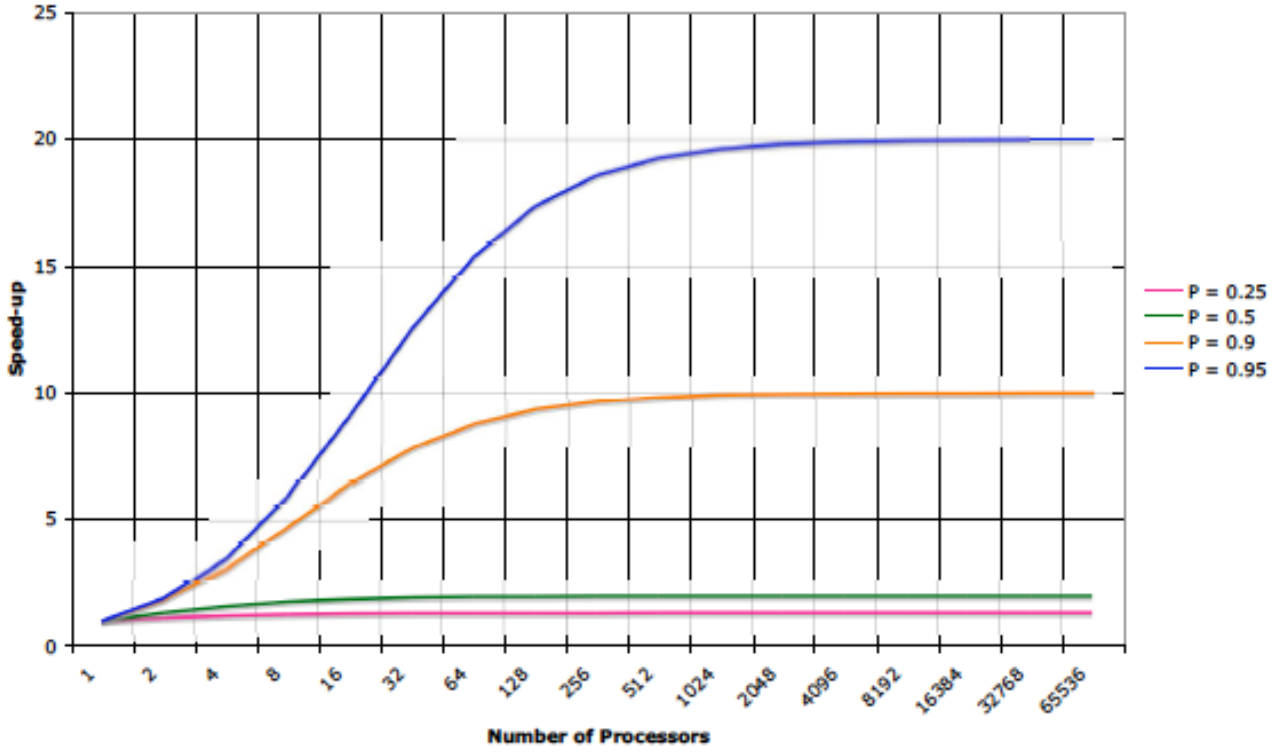
John Carmack (Doom, Quake, Oculus) : fast inverse square root using one iteration of Newton's method

```
float Q_rsqrt( float number ){
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y; // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 ); // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
    // y = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration, this can be removed

    #ifndef Q3_VM
    #ifdef __linux__
      assert( !isnan(y) ); // bk010122 - FPE?
    #endif
    #endif
    return y;
}
```

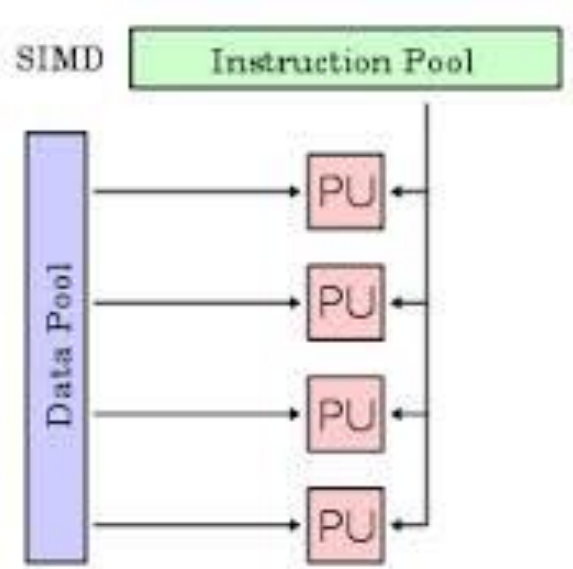
- Méthode de Newton
- Si on connaît une approximation x_n de la solution de l ’équation f(x) = 0
- On calcule la tangente à la courbe f(x) en x_n et on calcule son intersection avec l’axe Ox
- $y = f'(x_n)\ (x-x_n) + f(x_n) = 0$
- Cela nous donne une meilleure approximation
- $x_{n+1} = x_n - f(x_n)/f’(x_n)$

- Une nécessité d'utiliser toutes les capacités de calcul
  - CPU:
    - Multi-core
      - Hyperthreading (gain entre 15 et 30%)
    - SIMD Vecteurs
  - GPU
  - CPU many-core
- Mais une programmation complexe à mettre en œuvre.
- Différents langage, routines pour accélérer les calculs
- Une uniformisation avec OpenCL
  - API de calcul Open Source

- Programmation séquentielle ou parallèle ?
- Loi de Amdahl: S + P = 1, SU = 1/(S+P/N)



- Vecteur SIMD:
  - Single Instruction Multiple Data
  - Différents types
    - SSE -> 4*32 bits
    - AVX -> 8*32 bits
    - AVX2 ->16*32 bits



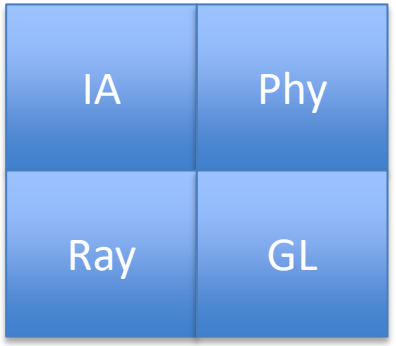
- Comment l'intégrer
  - A la main
  - Par auto vectorisation (dépend du compilateur)
  - En utilisant un modèle SPMD

- SPMD = Single Program Multiple Data
- ISPC est un compilateur C qui produit des applications compatible SIMD en programmant de manière séquentielle.
- Ce compilateur génère des binaires ou code source compatible SIMD.
- Cette méthode est assez proche de la programmation par kernel de OpenCL.

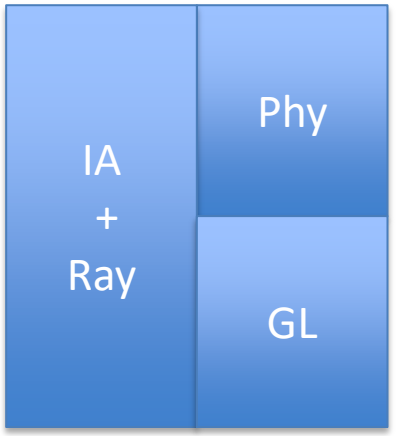
- Il est possible d'utiliser les fichiers objets avec GCC, ICC, …
- Supporte différentes architectures: SSE2, SSE4, AVX1, AVX2, Xeon Phi, …

```
export void simple(uniform float vin[], uniform float vout[],
                   uniform int count) {
    foreach (index = 0 ... count) {
      float v = vin[index];

      if (v < 3.)
          v = v * v;
       else
          v = sqrt(v);

      vout[index] = v;
    }
}
```

- Programmation par thread
  - Exécution de codes différents



  - Exécution du même code en parallèle



- Comment s'y prendre:
  - Fork/Join
    - Implémentation bas niveau
    - Ex: dans le cadre d'activité différents

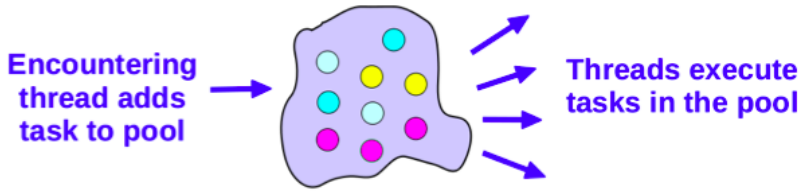
```
tid1 = fork(job1, a1);
job2(a2);
join tid1;
```

  - **Cobegin/Coend**
    - Méthode la plus simple
    - Méthode la plus utilisé
    - Limité à des boucle non imbriquées
    - Pas compatible avec la récursion

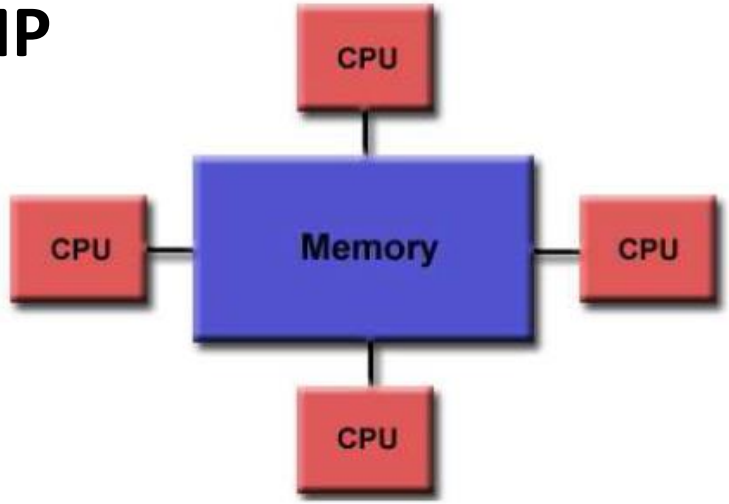
```
cobegin
        job1(a1);
        job2(a2);
coend
```

  - **Modèle par tâche**
    - Efficace pour des boucles non limités
    - Efficace pour la récursion

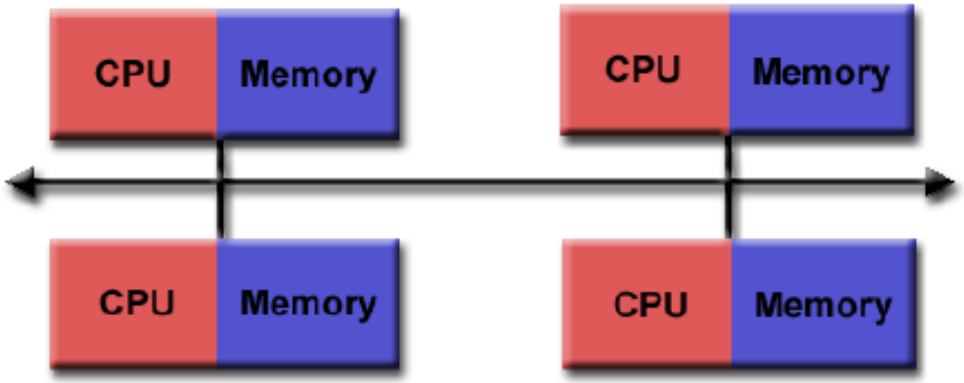
```
spawn(job1(a1));
spawn(job2(a2));
```



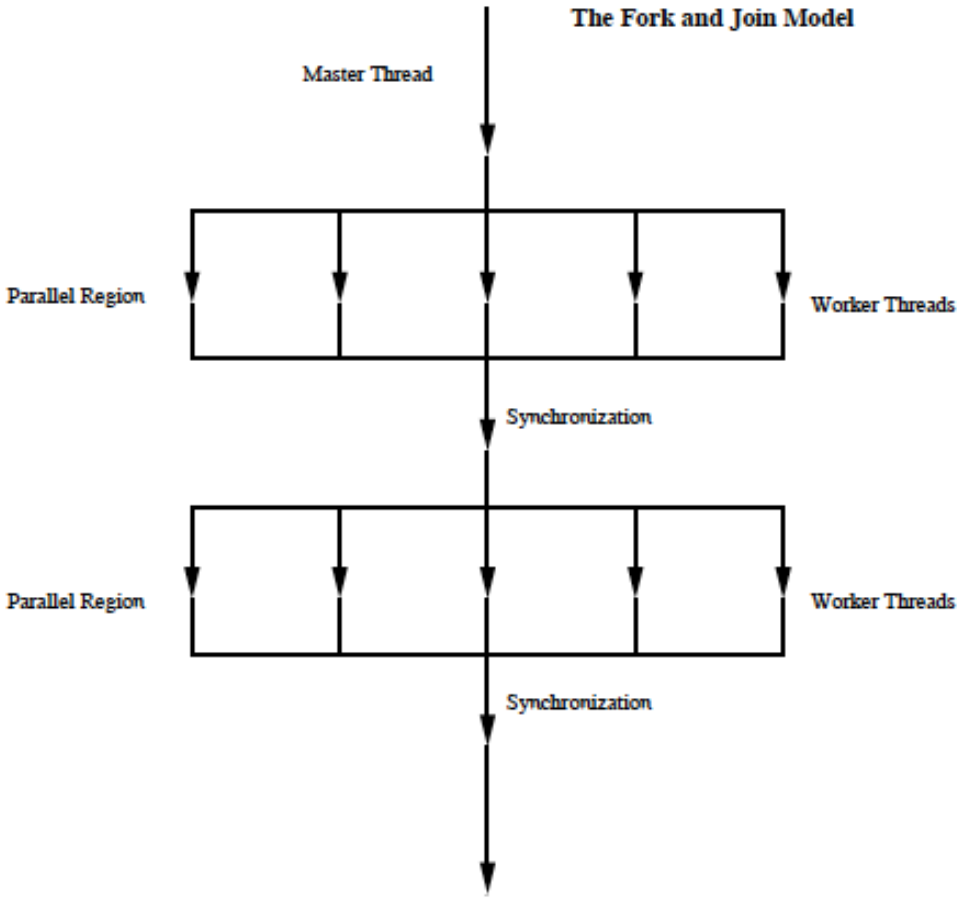
- Comment programmer en multiprocesseur
  - **Mémoire partagée : OpenMP**



  - Mémoire distribuée : MPI



- Principe d'OpenMP : Fork et Join



- Instructions cobegin/coend:
  - Pour paralléliser une région
    - #pragma omp parallel …
  - Différentes clauses
    - If (expression scalaire)
    - Num_thread (nombre)
    - Private (liste de variables)
    - Firstprivate (liste de variables)
    - Shared (liste de variables)
    - Default (liste de variables)

- OpenMP provides directives to support parallel loops.
- The full version:

```
#pragma omp parallel
#pragma omp for
    for (i = 0; i < n; i++)
        ...
```

- Abbreviated versions:

```
#pragma omp parallel for
    for (i = start; i < end; i++)
        ...
```

- There are some restrictions on the loop, including:
  - The loop has to be of this simple form with
    - `start` and `end` computable before the loop
    - a simple comparison test
    - a simple increment or decrement expression
  - exits with `break`, `goto`, or `return` are not allowed.

- Variables declared before a parallel block can be *shared* or *private*.
- Shared variables are shared among all threads.
- Private variables vary independently within threads
  - On entry, values of private variables are undefined.
  - On exit, values of private variables are undefined.
- By default,
  - all variables declared outside a parallel block are shared
  - except the loop index variable, which is private
- Variables declared in a parallel block are always private
- Variables can be explicitly declared shared or private.

Source : Luke Tierney, Brief introduction to OpenMP

- Programmation par tâches en OpenMP:
  - #pragma omp task [clause]

- Exemples de clauses:
  - If (expression scalaire)
  - Untied
  - Private (liste de variables)
  - Firstprivate (liste de variables)
  - Shared (liste de variables)
  - Default (liste de variables)

```c
#include <stdio.h>
#include <omp.h>
int fib(int n)
{
  int i, j;
  if (n<2)
    return n;
  else
    {
       #pragma omp task shared(i) firstprivate(n)
       i=fib(n-1);

       #pragma omp task shared(j) firstprivate(n)
       j=fib(n-2);

       #pragma omp taskwait
       return i+j;
    }
}

int main()
{
  int n = 10;

  omp_set_dynamic(0);
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
}
```

- Comment gérer les problèmes de synchronisation
  - Lock
  - Opérations atomiques

- Pour plus de détails, on peut consulter
  - Mattson & Meadows. A “Hands-on” Introduction to OpenMP.
  - Kiessling. An Introduction to Parallel Programming with OpenMP.

Stockage

- Gestion des IO disque
  - L'écriture sur le disque a un coup non négligeable.
  - Réserver cette étape pour les sauvegarde massive
  - Pour le faire en temps réel, réaliser l'écriture par un thread (pas le thread principal)
  - Structurer au mieux les données écrites
  - Préférer la sauvegarde binaire
  - Mettre en place un outil de gestion de ressource
- Il est strictement interdit d'utiliser les adresses absolues !

- La sérialisation des données
  - Méthode de stockage des objets
  - Principalement utilisé pour
    - le chargement
    - La sauvegarde des états
    - Les préférences
- Mais C++ n'offre pas de méthode de sérialisation
  - Soit développer ces méthodes
  - Soit utiliser des méthodes existantes

- Un outil de sérialisation doit contenir:
  - Une méthode de sauvegarde des instances
  - Une méthode de sélection de données
    - Plusieurs sauvegarde par exemple
  - Supporter la sauvegarder sous différents formats
    - Release (binaire) vs Debug (verbose)
  - Chargement de différents média
  - Contrôler la taille de la sauvegarde

Réseau

- Pourquoi communiquer entre clients ?
  - Permettre un mode multi-joueurs
  - Permettre un mode massivement multi-joueurs
  - Envoie des informations du master vers le client

- Communication entre les classes de deux applications

- Pour rappel:
  - Le réseau est orienté paquets
  - Un paquet est un petit ensemble d'information
  - Le chemin réseau n'est pas toujours le même
  - Les paquets n'arrivent pas toujours dans leur ordre initial

- De nombreuses méthodes de communications
  - Mais traditionnellement:
    - TCP
    - UDP

- The QThread class provides a platform-independent way to manage threads.
- A QThread object manages one thread of control within the program. QThreads begin executing in run(). By default, run() starts the event loop by calling exec() and runs a Qt event loop inside the thread.
- You can use worker objects by moving them to the thread using QObject::moveToThread().

- The QTcpServer class provides a TCP-based server.
- This class makes it possible to accept incoming TCP connections. You can specify the port or have QTcpServer pick one automatically. You can listen on a specific address or on all the machine's addresses.
- Call listen() to have the server listen for incoming connections. The newConnection() signal is then emitted each time a client connects to the server.
- If an error occurs, serverError() returns the type of error, and errorString() can be called to get a human readable description of what happened.
- Calling close() makes QTcpServer stop listening for incoming connections.

- QAbstractSocket is the base class for QTcpSocket and QUdpSocket and contains all common functionality of these two classes. If you need a socket, you have two options:
- TCP (Transmission Control Protocol) is a reliable, stream-oriented, connection-oriented transport protocol.
- UDP (User Datagram Protocol) is an unreliable, datagram-oriented, connectionless protocol.

- To open the socket, call connectToHost().
- At any time, QAbstractSocket has a state (returned by state()). The initial state is UnconnectedState. When the connection has been established, it enters ConnectedState and emitsconnected(). If an error occurs at any stage, error() is emitted.
- Read or write data by calling read() or write(), or use the convenience functions readLine() and readAll().
- QAbstractSocket also inherits getChar(), putChar(), and ungetChar() from QIODevice, which work on single bytes. The bytesWritten() signal is emitted when data has been written to the socket.
- To close the socket, call disconnectFromHost().

- Validation

- Il faut apprendre à débugger vos applications de manière efficace
  - Utiliser des outils de profiling
    - Ex: valgrind, gdb
  - Mettre en place des routines de tests
  - Mettre en place un mode debug et release

  - Détecter au plus tôt les erreurs
  - Adopter un modèle de programmation MVC

```
#include <iostream>

 using namespace std;

 int divint(int, int);

 int main() {
  int x = 5, y = 2;
  cout << divint(x, y);
  x =3; y = 0;
  cout << divint(x, y);
  return 0;
}

 int divint(int a, int b)
{
  return a / b;
}
```

g++ -g crash.cc -o crash

Lors de l'éexecution:

Floating point exception (core dumped)

```
gdb crash
   # Gdb prints summary information and then the (gdb) prompt

   (gdb) r
   Program received signal SIGFPE, Arithmetic exception.
   0x08048681 in divint(int, int) (a=3, b=0) at crash.cc:21
   21       return a / b;
   # 'r' runs the program inside the debugger
   # In this case the program crashed and gdb prints out some
   # relevant information.  In particular, it crashed trying
   # to execute line 21 of crash.cc.  The function parameters
   # 'a' and 'b' had values 3 and 0 respectively.

   (gdb) where
   #0  0x08048681 in divint(int, int) (a=3, b=0) at crash.cc:21
   #1  0x08048654 in main () at crash.cc:13
```

```
 (gdb) up
# Move from the default level '0' of
# the stack trace up one level
# to level 1.

   (gdb) list
# list now lists the code lines
# near line 13 of crash.cc

   (gdb) p x
# print the value of the local
# variable x in program main
```

```
# Equivalent to 'bt' or backtrace.  Produces what is known as a 'stack trace'.
# Read this as follows:  The crash occurred in the function divint at line 21 of crash.cc.
# This, in turn, was called from the function main at line 13 of crash.cc
```

- Dans les mini-projets, il faudra mettre en place des tests unitaires:
  - Procédure permettant de vérifier le bon fonctionnement d'une partie du code

- Des api pour vous aider …
  - cppTest
  - Qt
    - Par ex: QtTest

```
void TestQString::toUpper()
{
    QString str = "Hello";
    QCOMPARE(str.toUpper(), QString("HELLO"));
}
```

- Avant le TP ..

- Et ….
- Maintenant …

- Vous pouvez réaliser votre dernier commit pour le TP précédent.

TP