SCIENCES SORBONNE UNIVERSITÉ

Programmation Répartie
Master 1 Informatique – 4I400

Cours 9 : Concurrence 2
Promise, Future, Async

Yann Thierry-Mieg
Yann.Thierry-Mieg@lip6.fr

Plan

- On a vu au cours précédent :
- Protobuf, un format et un outillage pour la sérialisation
- Aujourd'hui : Concurrence le retour
  - Retour sur les outils thread : mutex, condition, atomic
  - Promise, Future
  - Async, packaged task

- Références :
- « Design Patterns », le GOF Gamma, Helm, Vlissides, Johnson
- « C++ concurrency in Action » Anthony Williams
- Slides assemblées de plusieurs sources, citées dans les slides concernées

En particulier Akim Demaille (EPITA) (pas mal de biblio citée)
https://www.lrde.epita.fr/~theo/lectures/CXXA/cxxa_5.pdf
https://www.lrde.epita.fr/~theo/lectures/CXXA/cxxa_6.pdf
Exemple BD : GlobalLogic Ukraine

Concurrence, Parallélisme

- Concurrence
  - Un Modèle de programmation
    - Différents flots de contrôle
  - Vise la clarté
    - Pas nécessairement l'efficacité
  - Certains langages entièrement basés sur la concurrence
    - GO, UrbiScript
- Parallélisme
  - Exécution simultanée de calculs
    - Mécanisme physique
  - Hyperthread, Multicore, GPU…

Mémoire

- Processus
  - Chacun son espace d'adressage
  - Communication uniquement via des canaux dédiés (IPC)
    - Signaux
    - Tubes
    - Memory Map
    - Sockets
- Threads
  - Partagent l'espace d'adressage
    - Lectures et écritures concurrentes possibles
    - Les variables de la pile (appels) sont locales au thread (en général)
  - Accès concurrents
    - Attention aux collisions,
      - mutex, atomic..

Thread vs Taches

- Multithreading
  - Création explicite de thread
    - Création de thread, affectation du travail, join
  - Latence due à la création/destruction des thread
    - Problème de passage à l'échelle, trop de thread tue la mule
- Multitask :
  - Partitionner le travail en morceaux parallélisables
    - Notion de tâche à réaliser
    - Parallélisable = concurrent, pas nécessairement parallèle
  - Laisse le runtime décider de l'allocation aux threads disponibles
    - Connaissance de la machine (compilateur, librairie)
  - Passe mieux à l'échelle (massivement multicore)
    - Parallélisme à grain plus fin => plus de parallélisme potentiel

Communiquer entre Thread

- Mémoire partagée
  - Les threads accèdent en concurrence aux données partagées
  - Nécessité de synchronisations
    - Atomic
    - Mutex
    - Condition variable
- Passage de messages
  - En principe pas de partage mémoire
    - Files de messages, channels, boite aux lettres…
  - Passe bien à l'échelle
    - Sur multi processus
    - Sur infrastructure distribuée
  - Mais plus lent que le partage mémoire direct
    - Copies, ou risque de réintroduction de data race.

Data Race

- Conflit d'accès
  - Deux (ou plus) threads accèdent en même temps au même emplacement mémoire
    - Au moins un d'entre eux est un écrivain
    - Résultats non définis
- Data Race
  - Un conflit d'accès non protégé
    - « les threads font la course »
    - Mauvaise idée, on ne contrôle pas qui gagnera
  - Synchronisations bloquantes pour corriger
    - Utilisation de mutex, lecteurs et écrivains partagent le lock
    - Possibilité de structures « lock free » cependant très difficiles à réaliser correctement

Modèle Mémoire

- Les différents processeurs
  - Ont une vision distincte de la mémoire
  - Chaque processeur a son propre cache
  - Les garanties pour la propagation entre caches sont relaxées
  - Des instructions dédiées (barrières mémoires) pour contrôler la cohérence
  - Accessibles en C++
- Les variables atomic
  - Disposent d'opération atomiques simples : incrément, compare and swap
  - Permettent l'introduction de barrières mémoire explicites
  - Sont la base de toutes les autres primitives de synchronisation

Absence de Data Race

- L'objectif est la cohérence « séquentielle »
  - Les langages/architectures modernes permettent de la contourner
  - Les raisonnements normaux s'écroulent

X=0 ; Y =0

(Thread A)	(Thread B)
X=1	Y=1
R1=Y	R2=X

Absence de Data Race

- L'objectif est la cohérence « séquentielle »
  - Les langages/architectures modernes permettent de la contourner
  - Les raisonnements normaux s'écroulent

X=0 ; Y =0

(Thread A)	(Thread B)
X=1	Y=1
R1=Y	R2=X

  - Ici on peut terminer avec : R1=R2=0
- L'absence de Data Race garantit la cohérence séquentielle
  - Donc toujours prévenir les conflits d'accès

Risques liés à la concurrence

- Nombre exponentiel d'entrelacements possibles
  - Si N processus indépendants, chacun pouvant prendre K états possibles, N^K états du programme entier
  - Difficile pour le raisonnement
  - Difficile à vérifier/contrôler pour les outils
- Erreurs et bugs de concurrence
  - Data race, violation de section critique, deadlocks, famines
  - Difficile à trouver
  - Difficile à reproduire (tests !)
  - Difficile à debugger

PROMISE, FUTURE

Motivation

```
// ...
device target_device;
target_device.initialize(); // <- time consuming

configuration_storage db;
configuration config;
db.load(config); // <- time consuming

target_device.configure(config);
```

Version thread

- Mise en place lourde
  - Échange de données limité (terminaison par join)

```
// ...
configuration_storage db;
configuration config;
thread loader([&db, &config]{ db.load(config); });
    // <- run loading config in a separate thread

device target_device;
target_device.initialize();
    // <- continue device initialization in current thread

loader.join();
    // <- assume loading done at this point

target_device.configure(config);
```

Pool de threads ?

```
class async_exec_service {
public:
    void exec(function<void()> f);
};
// ...
async_exec_service exec_service;
// ...
configuration_storage db;
configuration config;
exec_service.exec([&db, &config]{ db.load(config); });
    // <- run loading config in the thread pool
device target_device;
target_device.initialize();
    // <- continue device initialization in current thread

// hmmm... are we ready for proceed ???
target_device.configure(config);
```

Pool + Barrières

```
class async_exec_service { public: void exec(function<void()> f); };
// ...
async_exec_service exec_service;
// ...
configuration_storage db;
unique_ptr<configuration> config;

mutex config_guard;
condition_variable config_cv;
exec_service.exec([&]{
    unique_lock<mutex> lock(config_guard);
    config.reset(new configuration);
    db.load(*config);
    config_cv.notify_one();
}); // <- run loading config in the thread pool

device target_device;
target_device.initialize(); // continue device initialization in current thread
{ // wait for configuration loading done
    unique_lock<mutex> lock(config_guard);
    config_cv.wait(lock, [&]{ return (config != nullptr); });
}
// we are ready for proceed
target_device.configure(*config);
```

Programmation par Tâches

- Lancer un traitement puis obtenir sa réponse
  - Calculée par un autre thread
  - Directement en mémoire
  - Nécessité de synchronisation
- Comment communiquer le résultat de la tâche
  - Un canal de communication<T> standard pour échanger un résultat
    - Un état partagé, variable(s) accédées par les deux threads
    - **promise :** on doit la tenir, et écrire dedans le résultat
      - **Extrémité en écriture**
      - set_value(T)
    - **future :** on obtient la valeur au bout d'un moment
      - **Extrémité en lecture**
      - wait(), get()
  - Le **future** est bloquant tant que la **promise** n'est pas encore tenue

Promise/Future

```
void hello(std::promise<std::string>& prm)
{
 std::string res{"Hello from future"};
 prm.set_value(res);
}

int main()
{
 std::promise<std::string> prm;
 std::future<std::string> fut(prm.get_future());
 std::thread t(hello, std::ref(prm));
 std::cout <<"Hello from main\n";
 std::cout << fut.get() <<'\n';
 t.join();
}
```

Promise/ future

```
class async_exec_service {
public:
    template <typename _Function>
    auto exec(_Function&& f) -> future<decltype(f())>
    { /*...*/ }
};

async_exec_service exec_service;

// ...
configuration_storage db;
future<configuration> config = exec_service.exec([&db]{
    configuration config;
    db.load(config);
    return config;
});

device target_device;
target_device.initialize();

target_device.configure(config.get());
    // wait for result and proceed
```

Promise / Future

- Permet aussi de loger des exceptions
  - set_exception au lieu de set_value
  - Le get() va lever l'exception
  - Utile même en mono thread
- L'état partagé accessible via le future
  - Est consommé par get(), s'il n'est pas trivial
  - **shared_future** peut être construit à partir de **future**
  - Permet d'attendre à plusieurs une résultat
    - Invalide l'objet future sous jacent

Opérations du future

- valid()
  - Vrai si le future est encore utilisable (associé à un état partagé)
- get()
  - Rend la valeur associée
  - Bloquant si elle n'est pas disponible
- wait()/wait_for()/wait_until()
  - Attente bloquante ou avec timeout
- share(), ou simplement construire un shared_future<T>(f)
  - Construire une version partagée
  - Invalide le future

Taches : async

- L'exemple est encore assez lourd syntaxiquement

```
void hello(std::promise<std::string>& prm)
{
 std::string res{"Hello from future"};
 prm.set_value(res);
}

int main()
{
 std::promise<std::string> prm;
 std::future<std::string> fut(prm.get_future());
 std::thread t(hello, std::ref(prm));
 std::cout <<"Hello from main\n";
 std::cout << fut.get() <<'\n';
 t.join();
}
```

Async

- Du code parasite
  - Sur la fonction appelée
    - Signature prend un « promise<T>» au lieu de T
    - Signature rend void (forcé par thread)
    - Nécessaire de set_value explicite
  - Sur le code appelant
    - Création de thread
    - Join explicite
- Solution :
  - **async** : même signature que **thread**
  - Engendre automatiquement une paire promise/future
  - Encapsule les invocations pour avoir des signatures « normales »
  - Encapsule la création/fin de thread
  - Gère le degré de **parallélisme** indépendamment de la **concurrence**

Hello revisité

```
#include<future>
#include<iostream>

std::string hello()
{
 return"Hellofromfuture";
}

Int main()
{
 auto future =std::async(hello);
 std::cout <<"Hellofrommain\n";
 std::cout << future.get() <<'\n';
}
```

Exemple BD revisité

```
configuration_storage db;

future<configuration> config = async([&db]{
    configuration config;
    db.load(config);
    return config;
}); // run loading config in a separate thread

device target_device;
target_device.initialize();
    // continue device initialization in current thread

target_device.configure(config.get());
```

Politiques d'exécution

```
template< class Function, class... Args>
result_type async(Function&& f, Args&&... args);

template< class Function, class... Args >
result_type async(launch policy, Function&& f, Args&&... args);
```

- On peut spécifier une politique
  - **launch::async :** exécution dans un nouveau thread
  - **launch::defferred :** le future qui est retourné n'est pas concurrent
    - L'invocation à get() va faire calculer le résultat par le thread appelant
  - **launch::defferred | launch::async** : défaut, laisser le runtime décider

Politiques d'exécution : test

```
int working_thread_id()
{
  static std::unordered_map<std::thread::id, int> map;
  static std::mutex mapmut;

  std::this_thread::sleep_for(std::chrono::milliseconds{1});
  std::lock_guard<std::mutex> lock{mapmut};
  return
    map
    .emplace(std::this_thread::get_id(), map.size())
    .first
    ->second;
}
```

Politique défaut

```
std::vector<std::future<int>> futures;
for (size_t i = 0; i < 198; ++i)
  futures.emplace_back(std::async(1, working_thread_id));
for (size_t i = 0; i < futures.size(); ++i)
  std::cout << std::setw(3) << futures[i].get()
            << ((i+1) % 18 ? ' ' : '\n');
```

```
 2   1   3   4   5   6   7   8   9  11  12  10  13  14  15  16  17  18
19  20  21  22  23  25  24  26  27  28  29  30  31  32  33  34  35  36
37  38  40  39  41  42  43  44  45  46  47   2   1   3   4   5   6   7
 8   9  11  12  10  13  14  15  16  17  18  19  20  21  22  23  25  24
26  27  28  29  30  31  32  33  34  35  36   2  37   3   1   4   5   6
 7  38  40  39   8   9  11  12  10  13  14  15  16  17  18  19  20  21
22  23  25  24  26  27  28  41  29  30  31  32  33  42  34  35   2  36
 3   1   4  37   5   6   7   8   9  11  12  10  13  14  15  16  17  18
20  19  21  22  23  25  24  26  27  28  29  30  31  32  33  34  35  38
 2  36   3   1   4  37   5  40   6   8  39   9  41  11   7  12  10  13
14  42  43  44  18  20  19  45  46  15  16  17  21  22  23  25  24  26
```

Politique Deferred

With `std::async(std::launch::deferred, working_thread_id)`

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Politique async

- Sensiblement pareil que le défaut

With `std::async(std::launch::async, working_thread_id)`

2	1	3	4	5	6	7	8	9	11	12	10	13	14	15	16	17	18
19	20	21	22	23	25	24	26	27	28	29	30	31	32	33	34	35	36
37	38	40	39	41	42	2	1	3	4	5	6	7	8	9	11	12	10
13	14	15	16	17	18	19	20	21	22	23	25	24	26	27	28	29	30
31	32	33	2	1	3	4	5	6	7	8	9	11	12	10	13	14	15
34	35	36	16	17	18	19	20	37	38	21	22	23	25	40	24	26	27
28	29	30	31	39	32	2	1	3	4	5	6	33	41	7	9	8	11
42	12	10	13	14	15	34	16	17	18	19	20	21	22	23	25	24	26
27	28	29	35	30	31	32	36	37	2	1	38	3	4	5	6	33	7
8	9	11	40	12	10	13	39	14	15	16	17	18	34	19	41	20	42
21	22	23	25	24	26	27	28	29	35	30	31	32	36	2	1	37	3

Packaged_task

- Elément support pour la construction de async
  - Encapsule une fonction dans une paire future/promise
  - Pas de création de thread ou de concurrence
  - Construction similaire à async ou thread
    - Fonction + arguments en nombre et type arbitraire

Sort séquentiel

```
template <typename T>
std::list<T> quick_sort(std::list<T> input)
{
  if (input.empty())
    return input;
  auto res = std::list<T>{};
  res.splice(res.begin(), input, input.begin()); // Steal input[0].
  T const& pivot = res.front();
  auto divide_point = std::partition(input.begin(), input.end(),
                                     [&](T const& t){ return t<pivot; });
  auto lower_part = std::list<T>{};
  lower_part.splice(lower_part.end(), input, input.begin(), divide_point);
  auto new_lower = quick_sort(std::move(lower_part));
  auto new_higher = quick_sort(std::move(input));
  res.splice(res.end(), new_higher);
  res.splice(res.begin(), new_lower);
  return res;
}
```

Sort Concurrent

```
template <typename T>
std::list<T> quick_sort(std::list<T> input)
{
  if (input.empty())
    return input;
  auto res = std::list<T>{};
  res.splice(res.begin(), input, input.begin()); // Steal input[0].
  T const& pivot = res.front();
  auto divide_point = std::partition(input.begin(), input.end(),
                                     [&](T const& t){ return t<pivot; });
  auto lower_part = std::list<T>{};
  lower_part.splice(lower_part.end(), input, input.begin(), divide_point);
  auto new_lower = std::async(&quick_sort<T>, std::move(lower_part));
  auto new_higher = quick_sort(std::move(input));
  res.splice(res.end(), new_higher);
  res.splice(res.begin(), new_lower.get());
  return res;
}
```

Tris

```
int main(int argc, const char* argv[])
{
  size_t n = 1 < argc ? boost::lexical_cast<size_t>(argv[1]) : 4000;

  using ints = std::list<int>;
  auto l = ints(n); // *Not* ints{n}!
  std::generate(std::begin(l), std::end(l), std::rand);

  {
    auto res = ints{};
    chrono("sequential", n, [&]{ res = seq::quick_sort(l); });
    assert(std::is_sorted(std::begin(res), std::end(res)));
  }

  {
    auto res = ints{};
    chrono("parallel", n, [&]{ res = par::quick_sort(l); });
    assert(std::is_sorted(std::begin(res), std::end(res)));
  }
}
```

Bilan ?

- Pour 4000 entrées

parallel:	4000	456.60ms
sequential:	4000	3.04ms

- Attention au grain et au coût de mise en place
- Le coût est amorti par des jeux de données plus importants

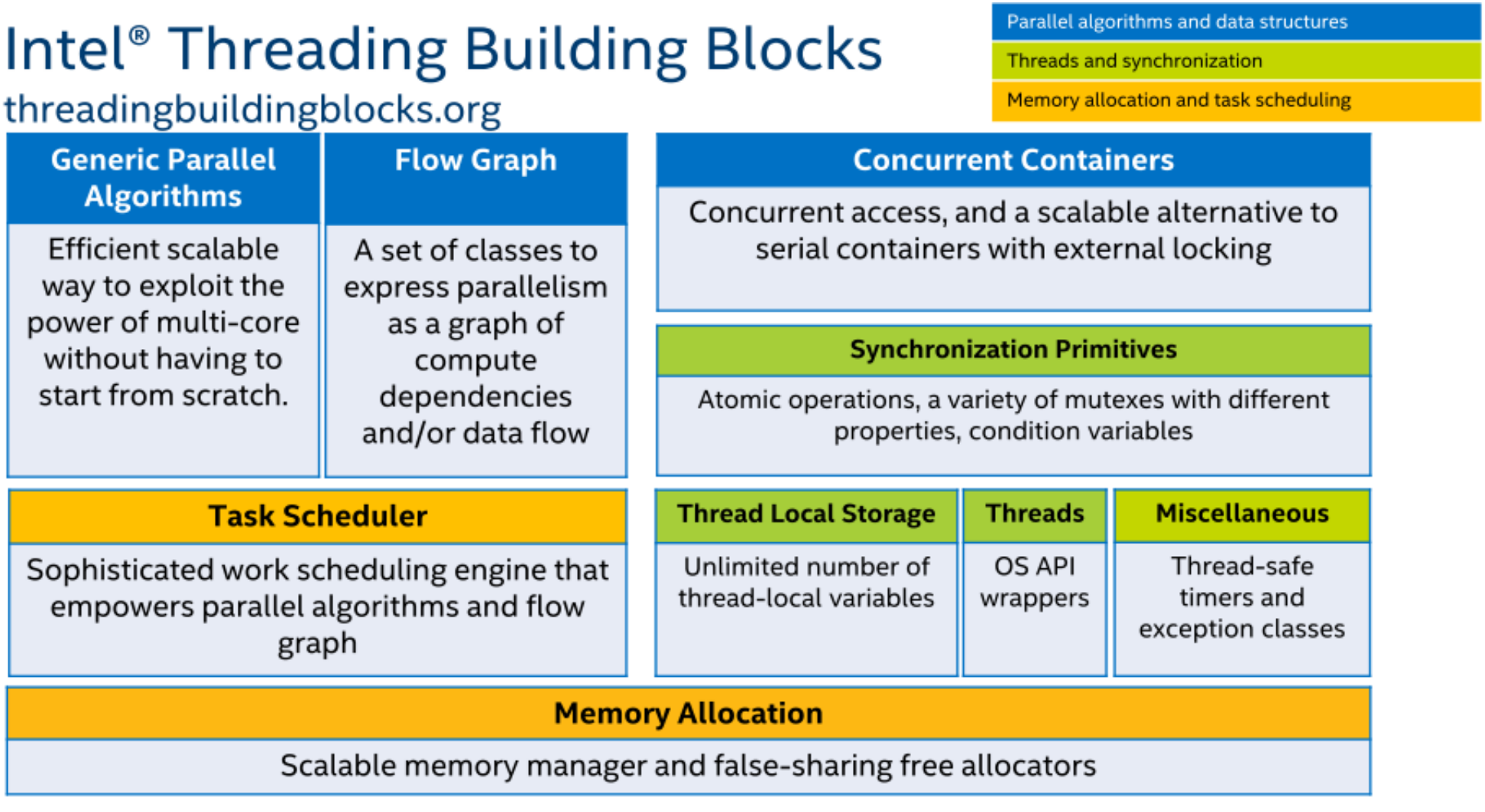
TASK BASED

Exemple : Intel TBB

Introduction à Intel TBB

- Librairie pour le parallélisme grain fin
  - Très utilisée, industriellement et académiquement
  - Très efficace et bien organisée
- Contient
  - Des algorithmes parallèles
  - Des structures de données concurrentes (lock free si possible)
  - Des primitives de synchronisation avancées
  - Gestion efficace des allocations mémoire concurrentes
  - Passage à l'échelle sur grand nombre de tâches, scheduling avancé
- Bénéfices
  - Bibliothèque C++, tous compilateurs, tous OS, protable
  - « icc » évidemment à privilégier + support architectures intel Atom, Core, Xeon … particulièrement bon

Vue d'ensemble



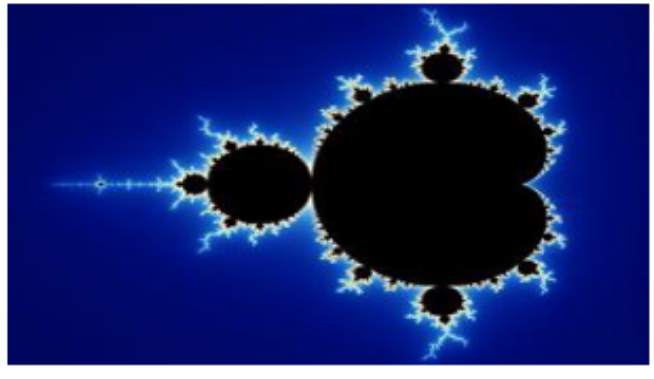
Exemple Mandelbrot

Mandelbrot Speedup

Intel® Threading Building Blocks (Intel® TBB)

```
int mandel(Complex c, int max_count) {
  int count = 0; Complex z = 0;
  for (int i = 0; i < max_count; i++) {
    if (abs(z) >= 2.0) break;
    z = z*z + c; count++;
  }
  return count;
}
```

Parallel algorithm



Task is a function object

```
parallel_for( 0, max_row,
  [&](int i) {
    for (int j = 0; j < max_col; j++)
      p[i][j]=mandel(Complex(scale(i),scale(j)),depth);
  }
);
```

Use C++ lambda functions to define function object in-line

Exemple QuickSort

```
template <typename RandomAccessIterator>
inline void parallel_sort(RandomAccessIterator begin,
                          RandomAccessIterator end)
{
  using value_type
    = typename std::iterator_traits<RandomAccessIterator>::value_type;
  parallel_sort(begin, end, std::less<value_type>());
}
```

- Version de base :
  - pose la comparaison par défaut
  - Trie de début à end

QuickSort Parallèle (2)

```
template <typename RandomAccessIterator, typename Compare>
void parallel_sort(RandomAccessIterator begin, RandomAccessIterator end,
                   const Compare& comp)
{
  if (begin < end)
    {
      const int min_parallel_size = 500;
      if (end - begin < min_parallel_size)
        std::sort(begin, end, comp);
      else
        internal::parallel_quick_sort(begin, end, comp);
    }
}
```

- Tester : taille > 500 ou on laisse tomber le parallélisme

Quicksort (3)

```
namespace internal
{
  // Wrapper method to initiate the sort by calling parallel_for.
  template <typename RandomAccessIter, typename Comp>
  void
  parallel_quick_sort(RandomAccessIter begin, RandomAccessIter end,
                      const Comp& comp)
  {
    tbb::parallel_for
      (quick_sort_range<RandomAccessIter, Comp>(begin, end-begin, comp),
       quick_sort_body<RandomAccessIter, Comp>(),
       auto_partitioner());
  }
}
```

- Cœur « internal » :
  - construire un « range »
  - Utiliser « parallel_for »
  - Lancer avec un « partitioner » (découpe)

QuickSort (4)

```
namespace internal
{
  // Sort elements in a range that is smaller than the grainsize.
  template <typename RandomAccessIterator, typename Compare>
  struct quick_sort_body
  {
    using range_t = quick_sort_range<RandomAccessIterator, Compare>;
    void operator()(const range_t& range) const
    {
      std::sort(range.begin, range.begin + range.size, range.comp);
    }
  };
}
```

- Tri des paquets d'éléments trop petits pour le parallélisme

QuickSort (5)

```
template <typename RandomAccessIterator, typename Compare>
class quick_sort_range: private no_assign
{
public:
  const Compare &comp;
  RandomAccessIterator begin;
  size_t size;

  quick_sort_range(RandomAccessIterator b, size_t s, const Compare &c)
    : comp(c), begin(b), size(s)
  {}

  bool empty() const { return size == 0; }
  bool is_divisible() const
  {
    static const size_t grainsize = 500;
    return grainsize <= size;
  }
};
```

- Classe support : un range, sans copie

QuickSort (6)

```
quick_sort_range(quick_sort_range& range, split) : comp(range.comp)
{
  RandomAccessIterator array = range.begin;
  if (size_t m = pseudo_median_of_9(array, range))
    std::swap(array[0], array[m]);
  RandomAccessIterator key0 = range.begin;

  // Really partition...
  // array[0..j) is less or equal to key.
  // array(j..s) is greater or equal to key.
  // array[j] is equal to key.

  begin = array + (j + 1);
  size = range.size - (j + 1);
  range.size = j;
}
```

- Tri + découpe

Résultat ?

```
using ints_t = std::vector<int>;
ints_t ints(n); // *Not* ints{n}!
std::generate(begin(ints), end(ints), std::rand);
{
  ints_t l{ints};
  chrono("std", n, [&]{ std::sort(begin(l), end(l)); });
  assert(std::is_sorted(begin(l), end(l)));
}
{
  ints_t l{ints};
  chrono("tbb", n, [&]{ mytbb::parallel_sort(begin(l), end(l)); });
  assert(std::is_sorted(begin(l), end(l)));
}
```

```
std: 10000000    802.49ms

tbb: 10000000    450.36ms
```

TBB : autres éléments

Basic algorithms `parallel_for`, `parallel_reduce`, `parallel_scan`

Advanced algorithms `parallel_while`, `parallel_do`, `parallel_pipeline`, `parallel_sort`

Containers `concurrent_queue`, `concurrent_priority_queue`, `concurrent_vector`, `concurrent_hash_map`

Scalable memory allocation `scalable_malloc`, `scalable_free`, `scalable_realloc`, `scalable_calloc`, `scalable_allocator`, `cache_aligned_allocator`

Mutual exclusion `mutex`, `spin_mutex`, `queuing_mutex`, `spin_rw_mutex`, `queuing_rw_mutex`, `recursive_mutex`

Atomic operations `fetch_and_add`, `fetch_and_increment`, `fetch_and_decrement`, `compare_and_swap`, `fetch_and_store`

Timing portable fine grained global time stamp

Task Scheduler direct access to control the creation and activation of tasks

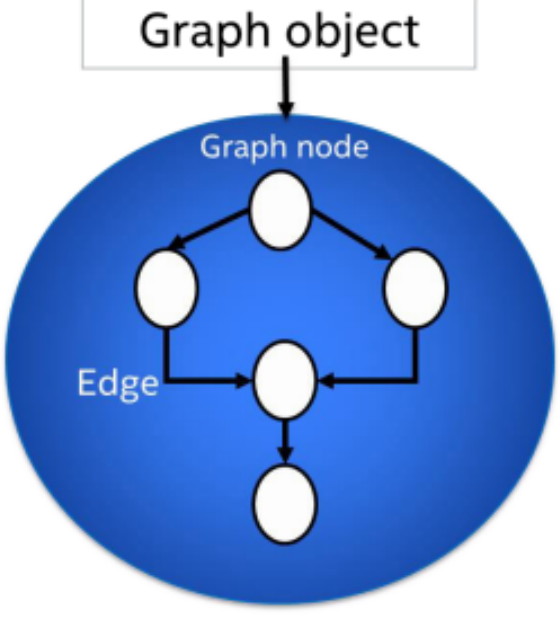
Taches, Flots

- Exprimer les dépendances entre tâches

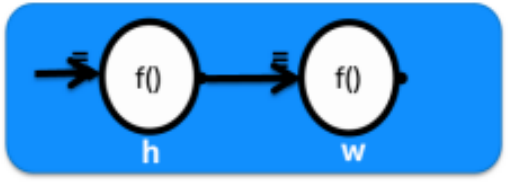
Efficient implementation of dependency graph and data flow algorithms

Design for shared memory application

Enables developers to exploit parallelism at higher levels

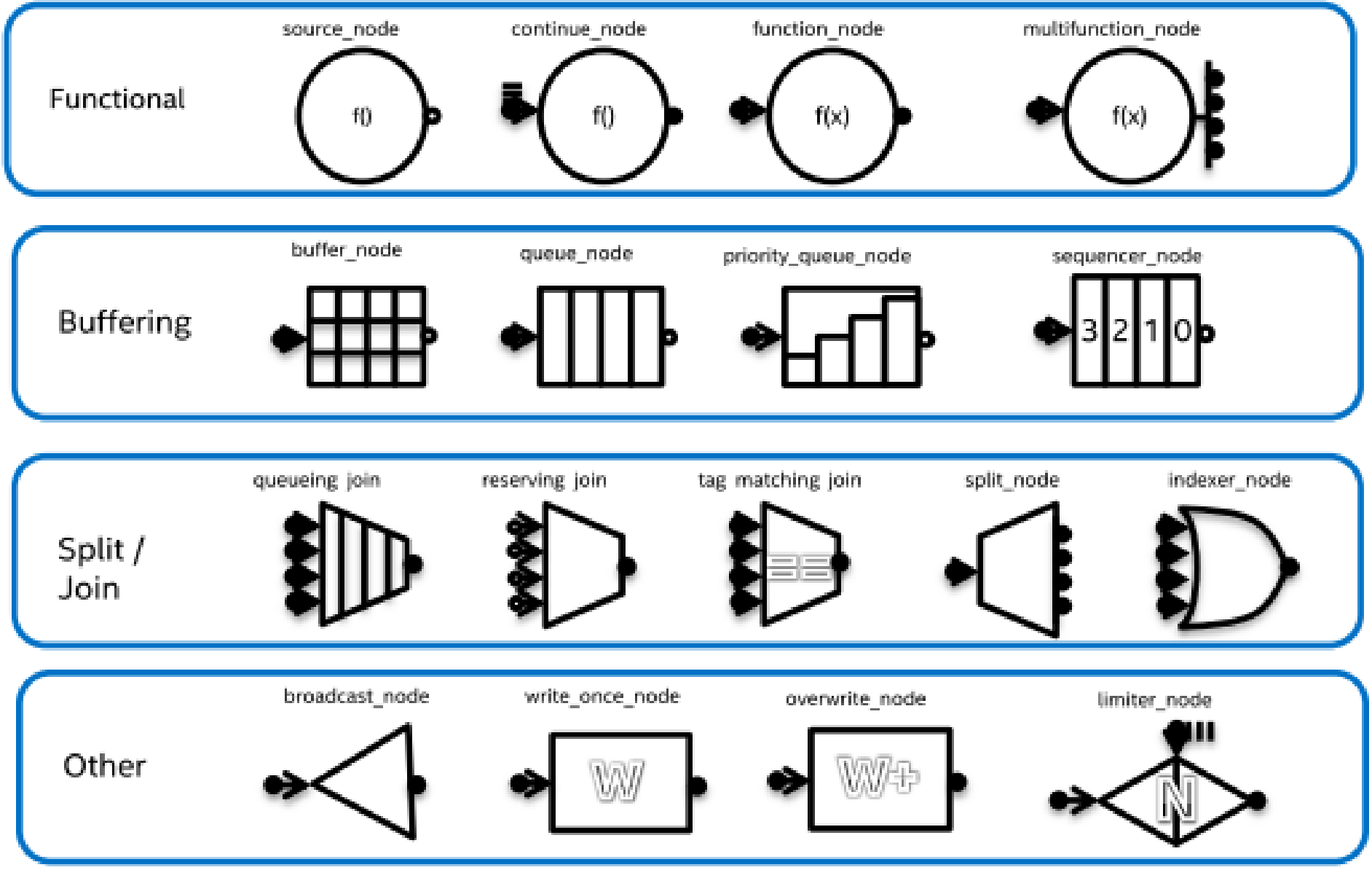


Hello World

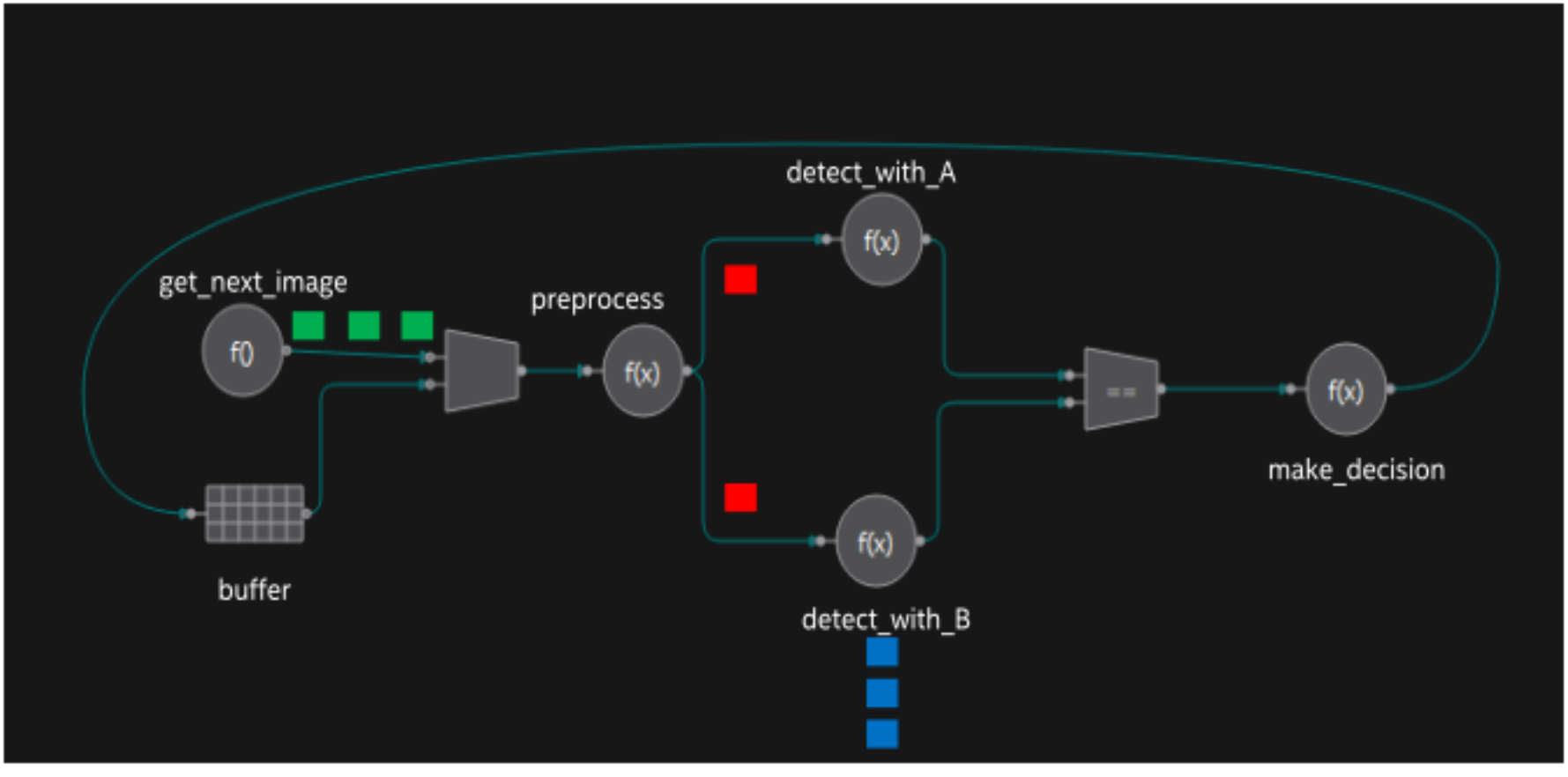



```
graph g;
continue_node< continue_msg > h( g,
    []( const continue_msg & ) {
        cout << "Hello ";
    } );
continue_node< continue_msg > w( g,
    []( const continue_msg & ) {
        cout << "World\n";
    } );
make_edge( h, w );
h.try_put(continue_msg());
g.wait_for_all();
```

Types de Noeuds

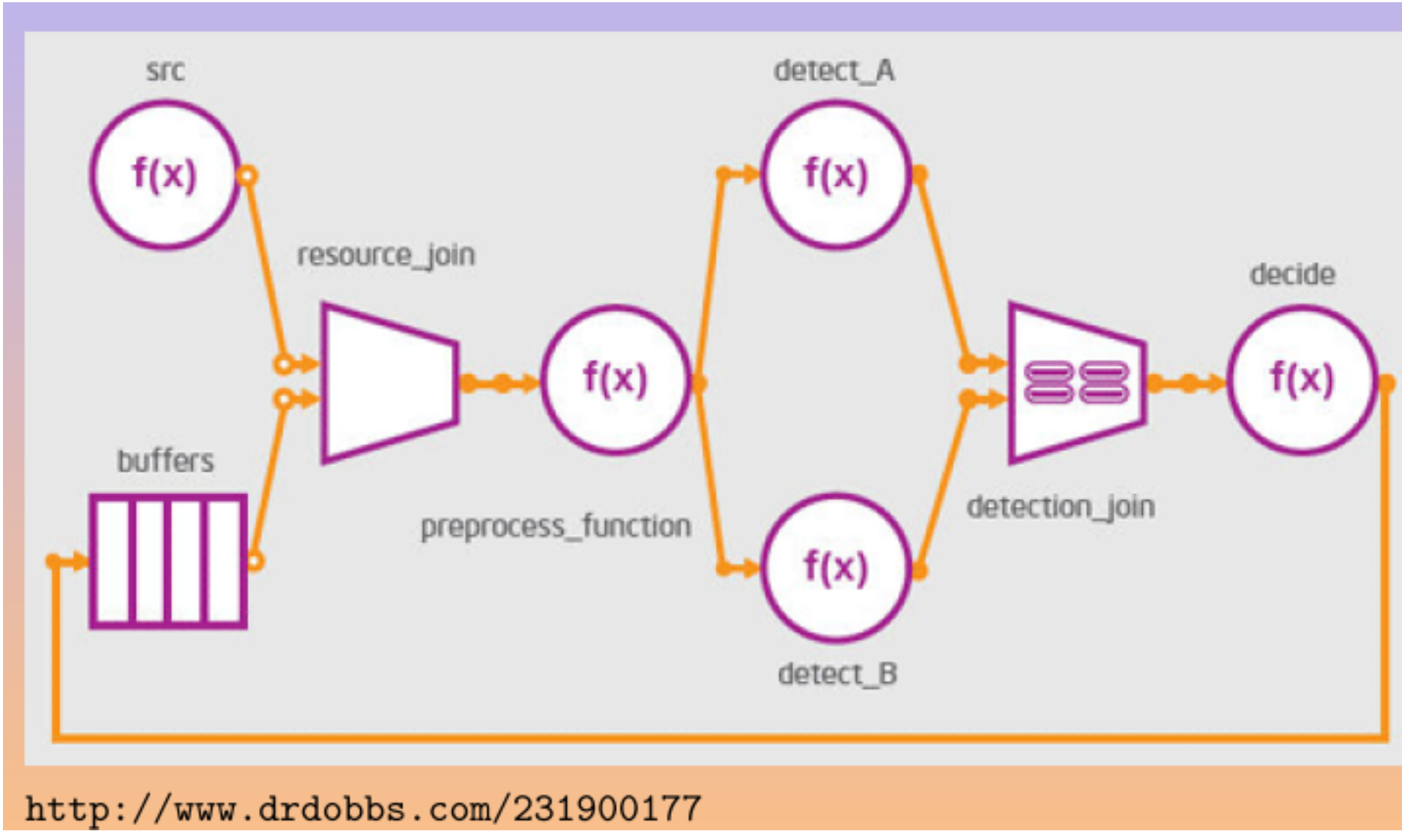


Algorithme = flow graph



Can express pipelining, task parallelism and data parallelism

Flow Graph (exemple 2)



http://www.drdobbs.com/231900177

Conclusion

- De nombreux autres features
  - Structures de données concurrentes e.g. lock free concurrent hash table
- L'approche parallélisme par tâches
  - Permet d'exprimer la concurrence de l'application
  - A niveau d'abstraction élevé
  - En retardant la réflexion sur le parallélisme effectif
- Plus la description est abstraite
  - Plus il y a de la place pour l'optimisation par les outils
    - A condition de leur faire confiance
  - Les outils e.g. TensorFlow très spécialisés
    - Très bonne « compilation » dédiée au hardware disponible
  - Les librairies génériques par tâche
    - Bon compromis, bonne base, n'exclut pas de devoir être plus fin