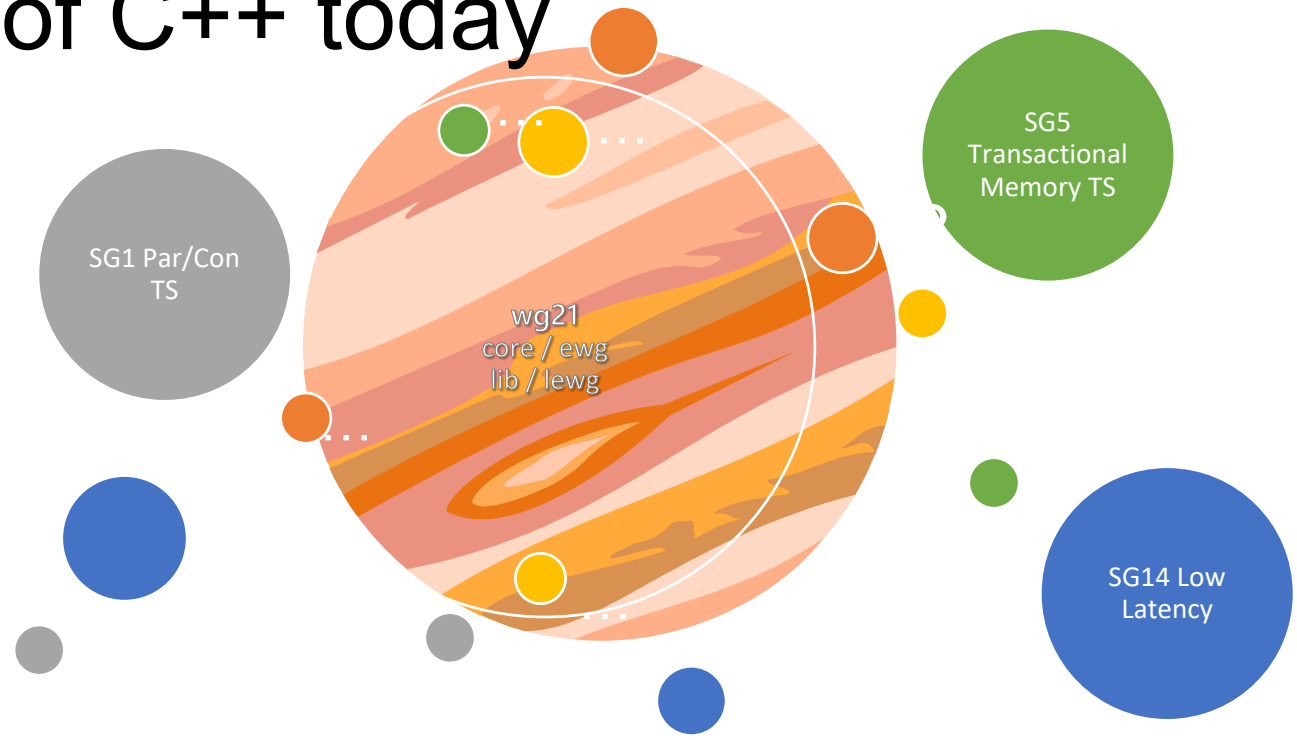
# The Parallel and concurrency planets of C++ today



# C++1Y(1Y=17/20/22) SG1/SG5/SG14 Plan red=C++17, blue=C++20? Black=future?

## Parallelism

- Parallel Algorithms:
- Data-Based Parallelism. (Vector, SIMD, ...)
- Task-based parallelism (cilk, OpenMP, fork-join)
- Execution Agents
- Progress guarantees
- MapReduce
- Pipelines

## Concurrency

- Future++ (then, wait_any, wait_all):
- Executors:
- Resumable Functions, await (with futures)
- Lock free techniques/Transactions
- Synchronics
- Atomic Views
- Co-routines
- Counters/Queues
- Concurrent Vector/Unordered Associative Containers
- Latches and Barriers
- upgrade_lock
- Atomic smart pointers

# Agenda

- Use the Proper abstractions?
- Why the rush to Massive Parallelism
- What Now?
- Hello World from C++11/14/17 Parallelism
- SYCL: C++ Heterogeneous (GPU) Programming
- Bonus: Executors
-

# “Hello World” with std::thread

```
#include <thread>
#include <iostream>

void func()
{
  std::cout << "**Inside thread "
   << std::this_thread::get_id() << "!" << std::endl;
}

int main()
{
  std::thread  t;
  t = std::thread( func );

  t.join();
  return 0;
}
```

*A simple function for thread to do…*

*Create and schedule thread…*

*Wait for thread to finish…*

# Avoiding errors / program termination…

```
#include <thread>
#include <iostream>

void func()
{
  std::cout << "**Hello world...\n";
}

int main()
{
  std::thread  t;
  t = std::thread( func );

  t.join();
  return 0;
}
```

*(1) Thread function must do **exception handling**; unhandled exceptions ==> termination…*

```
void func()
{
  try
  {
     // computation:
  }
  catch(...)
  {
     // do something:
  }
}
```

*(2) Must **join**, otherwise termination…*

***NOTE**: avoid use of detach( ) in C++11, difficult to use safely.*

# Example: saxpy


- Saxpy == *Scalar Alpha X Plus Y*
  - *Scalar multiplication and vector addition*

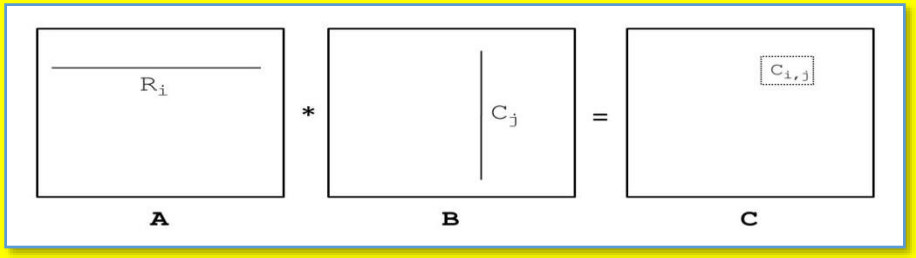
```
for (int i=0; i<n; i++)
   z[i] = a * x[i] + y[i];
```

**Parallel**

```
int start = …;
int end   = …;
for (int t=0; t<NumThreads; t++)
{
   thread(
     [&z,x,y,a,start,end]() -> void
     {
       for (int i = start; i < end; i++)
         z[i] = a * x[i] + y[i];
     }
   );

   start += …;
   end   += …;
}
```

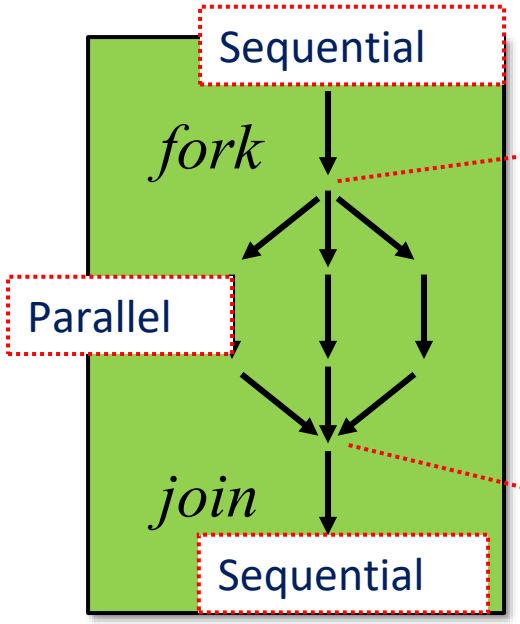
# Sequential Matrix Multiplication

```
//
// Naïve, triply-nested sequential solution:
//
for (int i = 0; i < N; i++)
{
   for (int j = 0; j < N; j++)
   {
      C[i][j] = 0.0;

      for (int k = 0; k < N; k++)
         C[i][j] += (A[i][k] * B[k][j]);
   }
}
```



# Structured ("fork-join") parallelism

- A common pattern when creating multiple threads



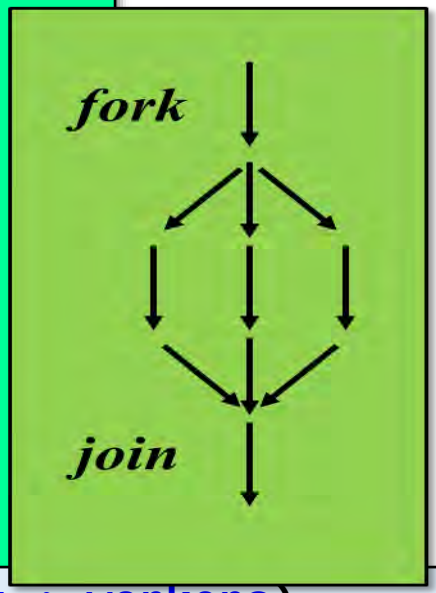

```
#include <vector>

std::vector<std::thread>  threads;

int cores = std::thread::hardware_concurrency();

for (int i=0; i<cores; ++i)   // 1 per core:
{
  auto code = []() { DoSomeWork(); };
  threads.push_back( thread(code) );
}
```

```
for (std::thread& t : threads) // new range-based for:
  t.join();
```

# Parallel solution

```
// 1 thread per core:
numthreads = thread::hardware_concurrency();
```

```
int rows  = N / numthreads;
int extra = N % numthreads;
int start = 0;      // each thread does [start..end)
int end   = rows;
vector<thread>  workers;
for (int t = 1; t <= numthreads; t++)
{
  if (t == numthreads) // last thread does extra rows:
    end += extra;
  workers.push_back( thread([start, end, N, &C, &A, &B](
  {
    for (int i = start; i < end; i++)
      for (int j = 0; j < N; j++)
      {
        C[i][j] = 0.0;
        for (int k = 0; k < N; k++)
          C[i][j] += (A[i][k] * B[k][j]);
      }
  }));

  start = end;
  end   = start + rows;
}
```




```
for (thread& t : workers)
   t.join();
```

*Going Parallel with C++11 by Joe Hummel*

# What does C++ Standard parallelism still need?

- Parallelism alone is not enough for the future…


**Expose parallelism**

**Maximize data locality:**
- **network**
- **disk**
- **RAM**
- **cache**
- **core**

**Minimize interaction:**
- **false sharing**
- **locking**
- **synchronization**



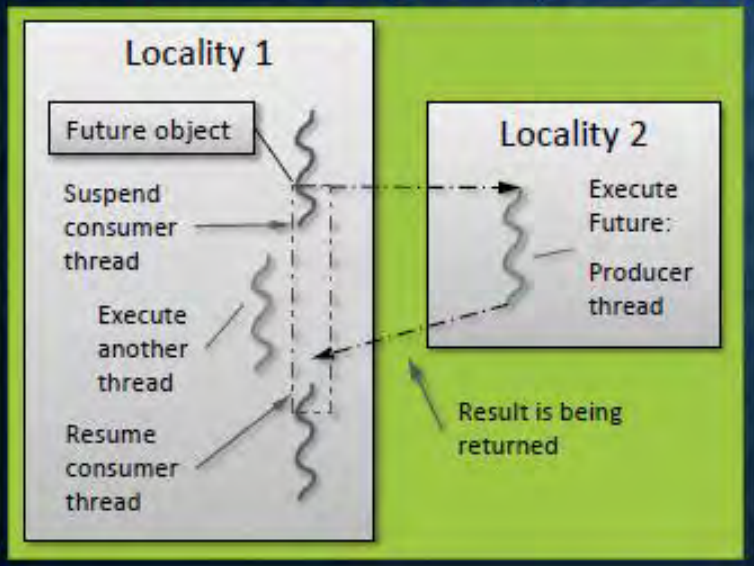
*Going Parallel with C++11 by Joe Hummel*

# Asynchronous Calls

- Building blocks:
  - std::async: Request asynchronous execution of a function.
  - Future: token representing function's result.
- Unlike raw use of std::thread objects:
  - Allows values or exceptions to be returned.
    - Just like "normal" function calls.

# WHAT IS A (THE) FUTURE

- A future is an object representing a result which has not been calculated yet




- Enables transparent synchronization with producer
- Hides notion of dealing with threads
- Makes asynchrony manageable
- Allows for composition of several asynchronous operations
- (Turns concurrency into parallelism)

# WHAT IS A (THE) FUTURE?

- Many ways to get hold of a future, simplest way is to use (std) async:

```
int universal_answer() { return 42; }

void deep_thought()
{
    future<int> promised_answer = async(&universal_answer);

    // do other things for 7.5 million years

    cout << promised_answer.get() << endl;   // prints 42, eventually
}
```

# WAYS TO CREATE A FUTURE

- Standard defines 3 possible ways to create a future,
  - 3 different '*asynchronous providers*'
    - std::async
      - See previous example, std::async has caveats
    - std::packaged_task
    - std::promise

# Standard Concurrency Interfaces

- std::async<>and std::future<>: concurrency as with sequential processing
  - one location calls a concurrent task and dealing with the outcome is as simple as with local sub-functions
- std: :thread: lOW-level approach
  - one location calls a concurrent lask and has to provide low-level techniques to handle the outcome
- std::promise<> and std::future<>: Simplify processing the outcome
  - one location calls a concurrent task but dealing with the outcome is simplified
- packaged_task<> : helper to separate task definition from call
  - one location defines a task and provides a handle for the outcome
  - another location decides when to call the task and the arguments
  - the call must not necessarily happen in another thread

# std::future Refresher

- *std::future<T>* -- a proxy for an eventual value of type *T*
- *std::promise<T>* -- a one-way channel to set the future.

# std::async + std::future

- Use async to start asynchronous operation
- Use returned future to wait upon result / exception

```
#include <future>

std::future<int> f = std::async( []() -> int
  {
    int result = PerformLongRunningOperation();
    return result;
  }
);
.
.
```

*START*

*lambda return type…*

*WAIT*

```
try
{
   int x = f.get();    // wait if necessary, harvest result:
   cout << x << endl;
}
catch(exception &e)
{
  cout << "**Exception: " << e.what() << endl;
}
```

# Async operations

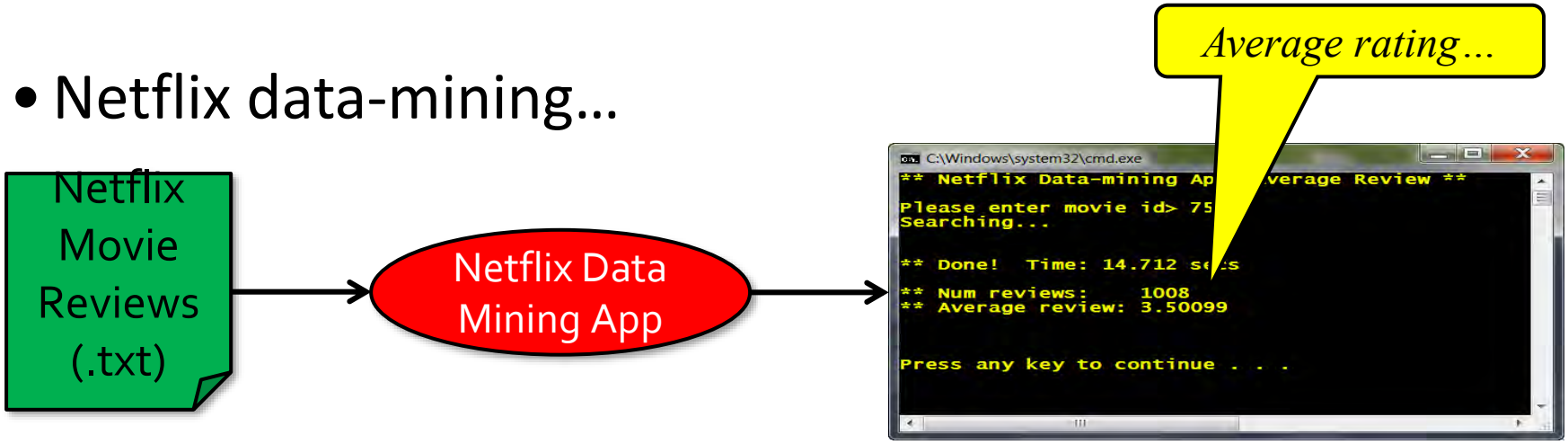
- Run on current thread *or* a new thread
- By default, system decides…
  - *based on current load, available cores, etc.*

```
// runs on current thread when you "get" value (i.e. lazy execution):
future<T> f1 = std::async( std::launch::deferred, []() -> T {...} );

// runs now on a new, dedicated thread:
future<T> f2 = std::async( std::launch::async, []() -> T {...} );

// let system decide (e.g. maybe you created enough work to keep system busy?):
future<T> f3 = std::async(↑[]() -> T {...} );
```

*optional argument missing*

# Commercial application

- Netflix data-mining…



# Sequential solution

```
cin >> movieID;

vector<string> ratings = readFile("ratings.txt");

tuple<int,int> results = dataMine(ratings, movieID);

int numRatings = std::get<0>(results);
int sumRatings = std::get<1>(results);
double avgRating = double(numRatings) / double(sumRatings);

cout << numRatings << endl;
cout << avgRating << endl;
```

```
dataMine(vector<string> &ratings, int id)
{
  foreach rating
    if ids match num++, sum += rating;

  return tuple<int,int>(num, sum);
}
```

# Parallel solution

```
dataMine(..., int begin, int end)
{
  foreach rating in begin..end
    if ids match num++, sum += rating;

  return tuple<int,int>(num, sum);
}
```

```
int chunksize = ratings.size() / numthreads;
int leftover  = ratings.size() % numthreads;
int begin     = 0;      // each thread does [start..end)
int end       = chunksize;

vector<future<tuple<int,int>>>  futures;

for (int t = 1; t <= numthreads; t++)
{
  if (t == numthreads)   // last thread does extra rows:
    end += leftover;

  futures.push_back(
    async([&ratings, movieID, begin, end]() ->
tuple<int,int>
    {
      return dataMine(ratings, movieID, begin, end);
    })
  );

  begin = end;
  end   = begin + chunksize;
}
```

```
for (future<tuple<int,int>> &f: futures)
{
   tuple<int, int> t = f.get();
   numRatings += std::get<0>(t);
   sumRatings += std::get<1>(t);
}
```