Introduction to parallelism in C++ with Threading Building Blocks

Andrea Bocci
CERN

set up

- the latest version of TBB is installed on the ESC machines under: `/opt/intel/oneapi/tbb/latest`
  - make it available to the compiler and to the program at run time:

```
$ source /opt/intel/oneapi/tbb/latest/env/vars.sh
```

  - check that it worked:

```
$ echo $TBBROOT
/opt/intel/oneapi/tbb/2022.2/env/..
```

- we need gcc 14 for some of the exercises
  - load the gcc 14 environment:

```
$ source scl_source enable gcc-toolset-14
$ gcc -version
gcc (GCC) 14.2.1 20250110 (Red Hat 14.2.1-7)
...
```

how to use TBB in your code

- include the TBB header in each file that uses TBB:

```
#include <tbb/tbb.h>
```

- tell the compiler where to find the TBB include files

```
g++ ... -I/opt/intel/oneapi/tbb/2022.3/include ...
```

- tell the compiler where to find the TBB library, and to the TBB library:

```
g++ ... -L/opt/intel/oneapi/tbb/2022.3/lib -ltbb ...
```

Note: the parts in yellow are not needed if you set up the library with

```
$ source /opt/intel/oneapi/tbb/latest/env/vars.sh
```

writing a Makefile

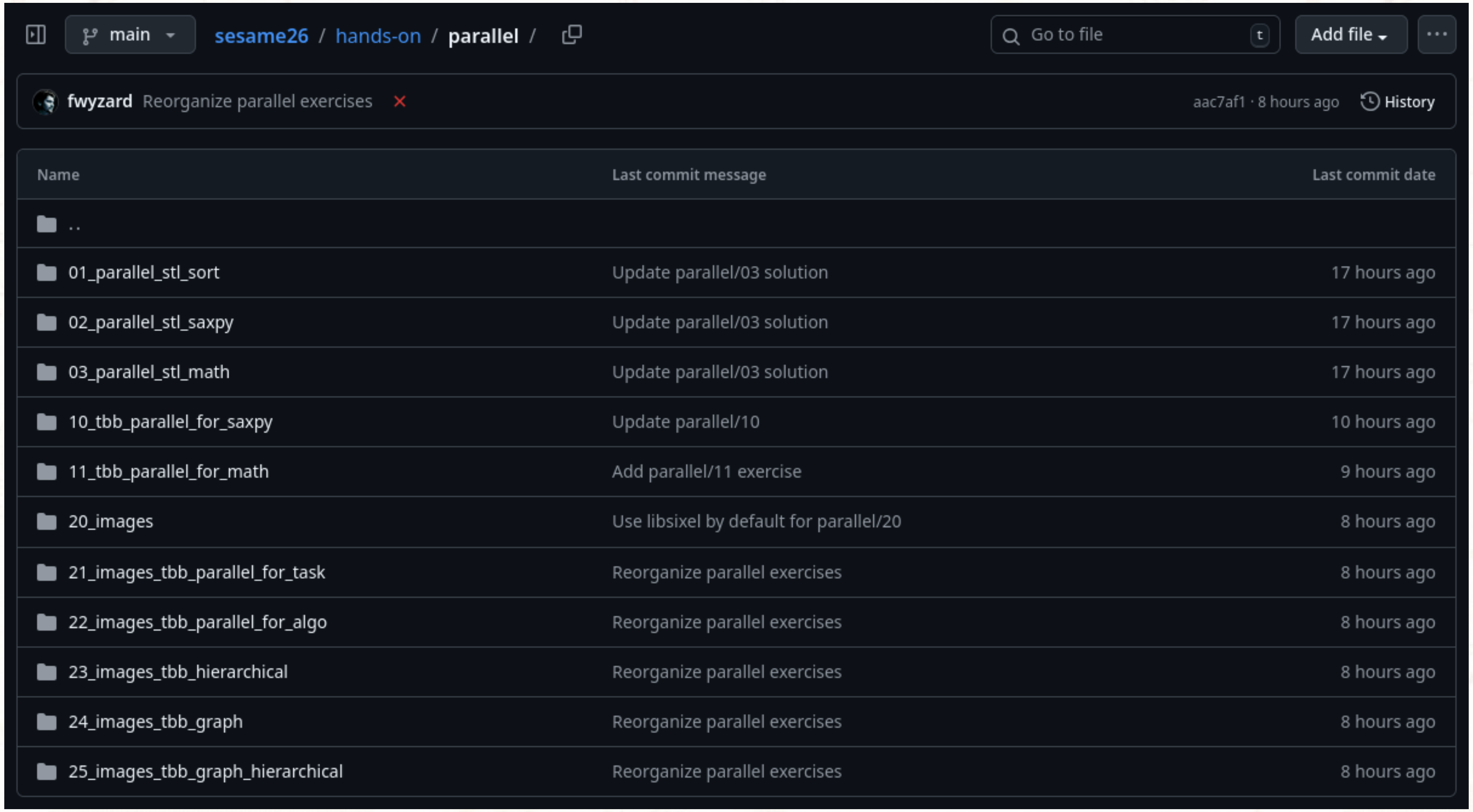
- a Makefile contains the recipe for building a program from its sources
  - simplifies building complex projects
  - builds only what is needed
  - can run multiple jobs in parallel

```
.PHONY: all clean

CXX      := g++
CXXFLAGS := -std=c++20 -O3 -g -Wall -march=native -I/opt/intel/oneapi/tbb/2022.3/include
LDFLAGS  := -L/opt/intel/oneapi/tbb/2022.3/lib -ltbb

all: test

clean:
	rm -f test

test: test.cc Makefile
	$(CXX) $(CXXFLAGS) $< $(LDFLAGS) -o $@
```

- an introduction to using Make: https://infn-esc.github.io/sesame26/basic/makefile.html
  - and in the GNU Make documentation

exercises on parallelism

hands-on/parallel/



fwyzard Reorganize parallel exercises ✗ aac7af1 · 8 hours ago

Name	Last commit message	Last commit date
..		
01_parallel_stl_sort	Update parallel/03 solution	17 hours ago
02_parallel_stl_saxpy	Update parallel/03 solution	17 hours ago
03_parallel_stl_math	Update parallel/03 solution	17 hours ago
10_tbb_parallel_for_saxpy	Update parallel/10	10 hours ago
11_tbb_parallel_for_math	Add parallel/11 exercise	9 hours ago
20_images	Use libsixel by default for parallel/20	8 hours ago
21_images_tbb_parallel_for_task	Reorganize parallel exercises	8 hours ago
22_images_tbb_parallel_for_algo	Reorganize parallel exercises	8 hours ago
23_images_tbb_hierarchical	Reorganize parallel exercises	8 hours ago
24_images_tbb_graph	Reorganize parallel exercises	8 hours ago
25_images_tbb_graph_hierarchical	Reorganize parallel exercises	8 hours ago

parallelism in C++ 11

parallelism in C++ 11

- C++ 11 introduced the building blocks to express parallelism in C++
  - threads:
    - `std::thread`
    - `std::jthread` (since C++20)
  - critical sections: “mutual exclusion”
    - `std::mutex`, …

    and locks:
    - `std::lock_guard`,
    - `std::scoped_lock` (since C++17), …
  - atomic operations:
    - `std::atomic<T>`,
    - `std::atomic_ref<T>` (since C++20)

parallelism in C++ 17

- C++ 17 introduced *parallel algorithms* and *execution policies* to express parallelism
  - `std::execution::sequenced_policy`
    - must not be parallelized
    - serial execution, same as the legacy algorithms
  - `std::execution::parallel_policy`
    - may be parallelized
    - may run in the calling thread or in other threads managed by the library
  - `std::execution::parallel_unsequenced_policy`
    - may be parallelized, vectorised, or migrated across threads
  - `std::execution::unsequenced_policy` (since C++20)
    - is not parallelized, but may be vectorised, *e.g.* with SSE, AVX2, AVX512, *etc.*

  - if you can express your problem using algorithms,
    parallel algorithms give you a simple way to leverage parallelism to speed up your code

algorithms: sorting

- sequential sorting
  - `std::sort(first, last)`
  - sort a container in place
  - traditional interface, no parallelism involved

algorithms: sorting

- sequential sorting
  - `std::sort(first, last)`
  - sort a container in place
  - traditional interface, no parallelism involved
- parallel sorting
  - `std::sort(policy, first, last)`
  - introduced in c++17, can speed up the sort algorithm using multiple threads
  - `policy can be`
    - `std::execution::seq`
    - `std::execution::unseq`
    - `std::execution::par`
    - `std::execution::par_unseq` (since c++20)

sorting random numbers

- hands-on/parallel/01_parallel_stl_sort/test.cc:
  - generate 1'000'000 random numbers

```
std::vector<std::uint64_t> v(size);
std::mt19937 gen{std::random_device{}()};
std::ranges::generate(v, gen);
```

  - sort them

```
std::sort(v.begin(), v.end());
```

  - check that they are sorted

```
assert(std::ranges::is_sorted(v));
```

measuring the execution time

- hands-on/parallel/01_parallel_stl_sort/test.cc:
  - measure how long that takes

```
const auto start = std::chrono::steady_clock::now();
std::sort(v.begin(), v.end());
const auto finish = std::chrono::steady_clock::now();
auto time =
  std::chrono::duration_cast<std::chrono::milliseconds>(finish - start).count();
std::cout << time << "ms\n";
assert(std::ranges::is_sorted(v));
```

  - wrap it all in a function, and repeat it a few times

```
void measure(std::vector<std::uint64_t> v) {
  ...
}

for (size_t i = 0; i < times; ++i) {
    measure(true, v);
}
```

the full exercise

- hands-on/parallel/01_parallel_stl_sort/test.cc:

```cpp
void measure(std::vector<std::uint64_t> v) {
  const auto start = std::chrono::steady_clock::now();
  std::sort(v.begin(), v.end());
  const auto finish = std::chrono::steady_clock::now();
  std::cout << std::chrono::duration_cast<std::chrono::milliseconds>(finish - start).count() << "ms\n";
  assert(std::ranges::is_sorted(v));
}

void repeat(std::vector<std::uint64_t> const& v, size_t times) {
  for (size_t i = 0; i < times; ++i) {
    measure(v);
  }
}

int main() {
  const std::size_t size = 1'000'000;
  const std::size_t repeats = 10;

  std::vector<std::uint64_t> v(size);
  std::mt19937 gen{std::random_device{}()};
  std::ranges::generate(v, gen);

  std::cout << "sequential sort\n";
  repeat(v, repeats);
  std::cout << '\n';
}
```

running the example

```
$ make
g++ -std=c++20 -O3 -g -Wall -march=native test.cc -o test

$ ./test
sequential sort
56ms
56ms
56ms
56ms
56ms
56ms
56ms
56ms
56ms
56ms
```

→ build and run the program on the ESC machine

→ change the sequential sort to use
`std::execution::seq`

→ try the other execution policies
`par`, `unseq`, `par_unseq`

running the example

```
$ make
g++ -std=c++20 -O3 -g -Wall -march=native test.cc -o test

$ ./test
sequential sort
56ms
56ms
56ms
56ms
56ms
56ms
56ms
56ms
56ms
56ms
```

→ you can find a working solution here

```
$ make
g++ -std=c++20 -O3 -g -Wall -march=native test.cc -ltbb -o test

$ ./test
std::execution::seq
55ms
55ms
55ms
55ms
...

std::execution::unseq
54ms
54ms
54ms
55ms
...

std::execution::par
8ms
3ms
2ms
2ms
...

std::execution::par_unseq
2ms
2ms
2ms
2ms
...
```

algorithms: arbitrary functions

- traditional `std::transform`
  - apply an unary (or binary) function to one (or two) input ranges
  - all operations are sequential
  - `std::transform(in_first, in_last, out_first, unary_op)`
  - `std::transform(in1_first, in1_last, in2_first, oit_first, binary_op)`

algorithms: arbitrary functions

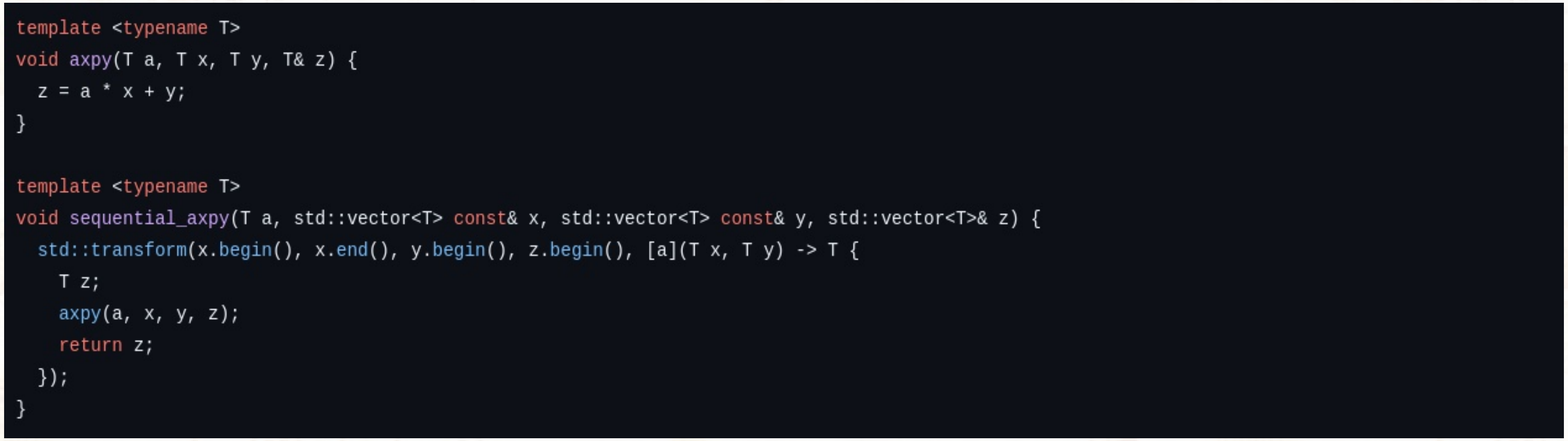
- traditional `std::transform`
  - apply an unary (or binary) function to one (or two) input ranges
  - all operations are sequential
  - `std::transform(in_first, in_last, out_first, unary_op)`
  - `std::transform(in1_first, in1_last, in2_first, oit_first, binary_op)`
- parallel `std::transform`
  - accepts an execution policy to (potentially) run the operations in parallel
  - `std::transform(policy, in_first, in_last, out_first, unary_op)`
  - `std::transform(policy, in1_first, in1_last, in2_first, oit_first, binary_op)`

algorithms: arbitrary functions

- traditional `std::transform`
  - apply an unary (or binary) function to one (or two) input ranges
  - all operations are sequential
  - `std::transform(in_first, in_last, out_first, unary_op)`
  - `std::transform(in1_first, in1_last, in2_first, oit_first, binary_op)`
- parallel `std::transform`
  - accepts an execution policy to (potentially) run the operations in parallel
  - `std::transform(policy, in_first, in_last, out_first, unary_op)`
  - `std::transform(policy, in1_first, in1_last, in2_first, oit_first, binary_op)`
- std::for_each
  - similar
  - accepts a unary function, does not return anything
  - the parallel version guarantees that the input collection is not copied

parallel transformation

- `hands-on/parallel/02_parallel_stl_saxpy/test.cc`:
  - generate a random scalar number `a`
  - generate two vectors of 100’000’000 random numbers `X` and `Y`
  - measure how log it takes to apply the “saxpy” kernel to the vectors
    - (single precision) `a x + y`



```cpp
template <typename T>
void axpy(T a, T x, T y, T& z) {
  z = a * x + y;
}

template <typename T>
void sequential_axpy(T a, std::vector<T> const& x, std::vector<T> const& y, std::vector<T>& z) {
  std::transform(x.begin(), x.end(), y.begin(), z.begin(), [a](T x, T y) -> T {
    T z;
    axpy(a, x, y, z);
    return z;
  });
}
```

  - use the parallel STL to speed up the operations

running the example

```
$ make
g++ -std=c++20 -O3 -g -Wall -march=native test.cc -o test

$ ./test
sequential saxpy
  29.9 ms
  29.9 ms
  29.9 ms
  29.9 ms
  30.5 ms
  29.9 ms
  29.8 ms
  30.4 ms
  29.9 ms
  30.5 ms
```

→ build and run the program on the ESC machine

→ change the sequential sort to use
`std::execution::seq`

→ try the other execution policies
`par`, `unseq`, `par_unseq`

running the example

```
$ make
g++ -std=c++20 -O3 -g -Wall -march=native test.cc -o test

$ ./test
sequential saxpy
  29.9 ms
  29.9 ms
  29.9 ms
  29.9 ms
  30.5 ms
  29.9 ms
  29.8 ms
  30.4 ms
  29.9 ms
  30.5 ms
```

→ you can find a working solution here

```
$ make
g++ -std=c++20 -O3 -g -Wall -march=native test.cc -ltbb -o test

$ ./test
std::execution::seq
  28.7 ms
  28.7 ms
  28.7 ms
  29.0 ms
...

std::execution::unseq
  28.7 ms
  28.7 ms
  28.7 ms
  29.0 ms
...

std::execution::par
   9.3 ms
   6.2 ms
   6.2 ms
   4.5 ms
...

std::execution::par_unseq
   4.5 ms
   4.5 ms
   4.7 ms
   4.5 ms
...
```

a more complex operation

- the “saxpy” example shows a relatively small speed-up, despite running on about one hundred CPU cores
- what can be the limiting factor ?
- how does that change with a more complex operation ?

```
template <typename T>
T kernel(T x, T y) {
  auto a = std::sin(x);
  auto b = std::cos(y);
  auto c = std::sqrt(a * a - b * b);
  return c;
}
```

a more complex operation

- the “saxpy” example shows a relatively small speed-up, despite running on about one hundred CPU cores
- what can be the limiting factor ?
- how does that change with a more complex operation ?

```cpp
template <typename T>
T kernel(T x, T y) {
  auto a = std::sin(x);
  auto b = std::cos(y);
  auto c = std::sqrt(a * a - b * b);
  return c;
}
```

→ you can find a working solution here

```
$ make
g++ -std=c++20 -O3 -g -Wall -march=native test.cc -ltbb -o test

$ ./test
std::execution::seq
 229.7 ms
 228.8 ms
 228.4 ms
 228.0 ms
...

std::execution::unseq
 229.1 ms
 229.5 ms
 229.3 ms
 229.3 ms
...

std::execution::par
  12.2 ms
   8.7 ms
   1.7 ms
   2.1 ms
...

std::execution::par_unseq
   1.7 ms
   1.6 ms
   2.6 ms
   1.7 ms
...
```

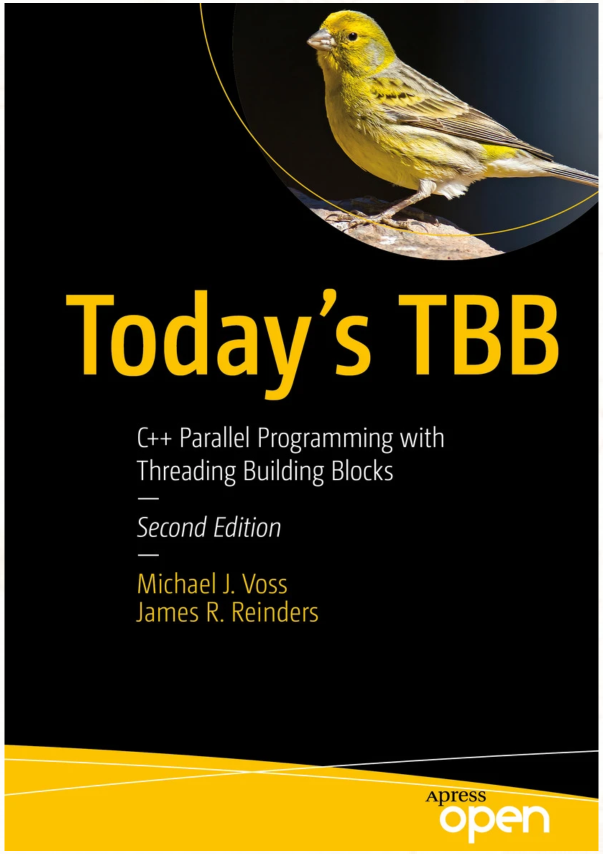
parallelism with Threading Building Blocks

parallelism with Threading Building Blocks

- Threading Building Blocks
  - developed by Intel as an open source project
  - now part of the oneAPI ecosystem
  - including the official documentation and reference

- why TBB ?
  - scalability and load balancing
  - composability
  - multiple levels of parallelism
    - task-based parallelism: `parallel_invoke`, `parallel_pipeline`, **various graph types**
    - fork-join parallelism: **`parallel_for`**, various parallel algorithms
  - access to low level interface
    - task_group, task_arena, observers, *etc.*

Today’s TBB



Today's TBB (2025)

C++ Parallel Programming with Threading Building Blocks

Second Edition

- by Michael J. Voss and James R. Reinders
- open access book: https://doi.org/10.1007/979-8-8688-1270-5

all examples in the book are on GitHub

- https://github.com/Apress/pro-TBB
- make sure to use the `Todays-TBB-2025` branch

tbb::parallel_for

- **tbb::parallel_for** is the most common parallel algorithm

```
template<typename Index, typename Func>
void parallel_for(Index first, Index last, Index step, const Func& func);
```

  - `step` is optional, defaults to 1
  - `func` can be a lambda !

- replace

```
for (int i = first; i < last; i += step) {
    // … loop body …
}
```

- with

```
tbb::parallel_for<int>(first, last, step, [&](int i){
    // … loop body …
});
```

“saxpy” with TBB

- hands-on/parallel/10_tbb_parallel_for_saxpy/test.cc:
  - similar exercise to to the one we have done with the parallel std::transform
  - this time, parallelise the loop using tbb::parallel_for

```
template <typename T>
void axpy(T a, T x, T y, T& z) {
  z = a * x + y;
}

template <typename T>
void sequential_axpy(T a, std::vector<T> const& x, std::vector<T> const& y, std::vector<T>& z) {
  std::size_t size = x.size();
  for (std::size_t i = 0; i < size; ++i) {
    axpy(a, x[i], y[i], z[i]);
  }
}
```

running the example

```
$ make
g++ -std=c++20 -O3 -g -Wall -march=native test.cc -o test

$ ./test
sequential saxpy
  28.9 ms
  28.9 ms
  29.0 ms
  28.9 ms
  29.1 ms
  28.9 ms
  28.9 ms
  29.0 ms
  28.9 ms
  29.0 ms
```

→ build and run the program on the ESC machine

→ change the sequential sort to use
  `tbb::parallel_for`

running the example

```
$ make
g++ -std=c++20 -O3 -g -Wall -march=native test.cc -o test

$ ./test
sequential saxpy
  28.9 ms
  28.9 ms
  29.0 ms
  28.9 ms
  29.1 ms
  28.9 ms
  28.9 ms
  29.0 ms
  28.9 ms
  29.0 ms
```

→ you can find a working solution here

```
$ make
g++ -std=c++20 -O3 -g -Wall -march=native test.cc -ltbb -o test

$ ./test
sequential saxpy
  29.3 ms
  28.9 ms
  28.9 ms
  29.2 ms
  28.9 ms
  29.2 ms
  28.8 ms
  28.9 ms
  29.3 ms
  28.9 ms

parallel saxpy
  12.2 ms
   6.4 ms
   4.8 ms
   4.5 ms
   4.5 ms
   7.7 ms
   4.8 ms
   4.4 ms
   4.5 ms
   4.4 ms
```

a more complex operation with TBB

- let's repeat the exercise with a more complex computation, using TBB

```
template <typename T>
T kernel(T x, T y) {
  auto a = std::sin(x);
  auto b = std::cos(y);
  auto c = std::sqrt(a * a - b * b);
  return c;
}
```

a more complex operation with TBB

- let's repeat the exercise with a more complex computation, using TBB

```
template <typename T>
T kernel(T x, T y) {
  auto a = std::sin(x);
  auto b = std::cos(y);
  auto c = std::sqrt(a * a - b * b);
  return c;
}
```

→ you can find a working solution here

```
$ make
g++ -std=c++20 -O3 -g -Wall -march=native test.cc -ltbb -o test

$ ./test
sequential computation
 227.8 ms
 227.6 ms
 227.3 ms
 227.7 ms
 228.5 ms
 227.7 ms
 228.3 ms
 228.0 ms
 228.3 ms
 228.0 ms

parallel computation
  12.0 ms
   8.8 ms
   3.6 ms
   1.8 ms
   1.7 ms
   1.7 ms
   1.8 ms
   1.7 ms
   1.7 ms
   1.7 ms
```

tbb::parallel_for partitioners

- `tbb::parallel_for`
  - splits the input range in *chunks*
  - executes the loop body over each chunk in parallel
- we can configure how TBB splits the work providing a *partitioner*

```
template<typename Index, typename Func>
void parallel_for(Index first, Index last, Index step,
                  const auto& partitioner,
                  const Func& func);
```

tbb::parallel_for partitioners

- the *partitioner* specifies a strategy for splitting into chunks and executing the loop:
  - `tbb::auto_partitioner` (default)
    - initially split the work in chunks of approximately equal size, trying to keep all threads busy
    - may split the work further if some items take longer than others
  - `tbb::static_partitioner`
    - distribute the work uniformly across threads, with no further splitting
    - may be more efficient if all items take approximately the same time
  - `tbb::affinity_partitioner`
    - similar to the `tbb::auto_partitioner`, but tries to maintain cache affinity across multiple loops
    - may be useful for adaptive algorithms that do multiple passes over the same data
  - `tbb::simple_partitioner`
    - split the input range as much as possible
    - useful with a `tbb::blocked_range` to process 1-, 2-, 3- or N-dimensional chunks

tbb::blocked_range

- `tbb::blocked_range(first, last, grainsize)`
  - specifies a semi-open range [`first`, `last`) that can be iteratively subdivided by TBB
  - `grainsize` specifies the smallest number of elements to keep in a single *chunk*
  - can be passed as input to tbb::parallel_for instead of (`first`, `last`, `step`, …):

```
tbb::blocked_range range(first, last, minsize);

tbb::parallel_for(range, [&](Index i){
    // … loop body …
});
```

- most useful in its 2-dimensional, 3-dimensional, and N-dimensional variants:

```
tbb::blocked_range2d range(row_begin, row_end, row_grainsize,
                           col_begin, col_end, col_grainsize);

tbb::blocked_range3d range(...);

tbb::blocked_nd_range range(...);
```

different partitioners

- try to use the various partitioners in the “saxpy” `tbb::parallel_for` exercise
    - what gives the best performance ?
    - what is a good chunk size for the `simple_partitioner` ?
    - does the `affinity_partitioner` make sense in this case ?

- how do things change with more complex computations ?

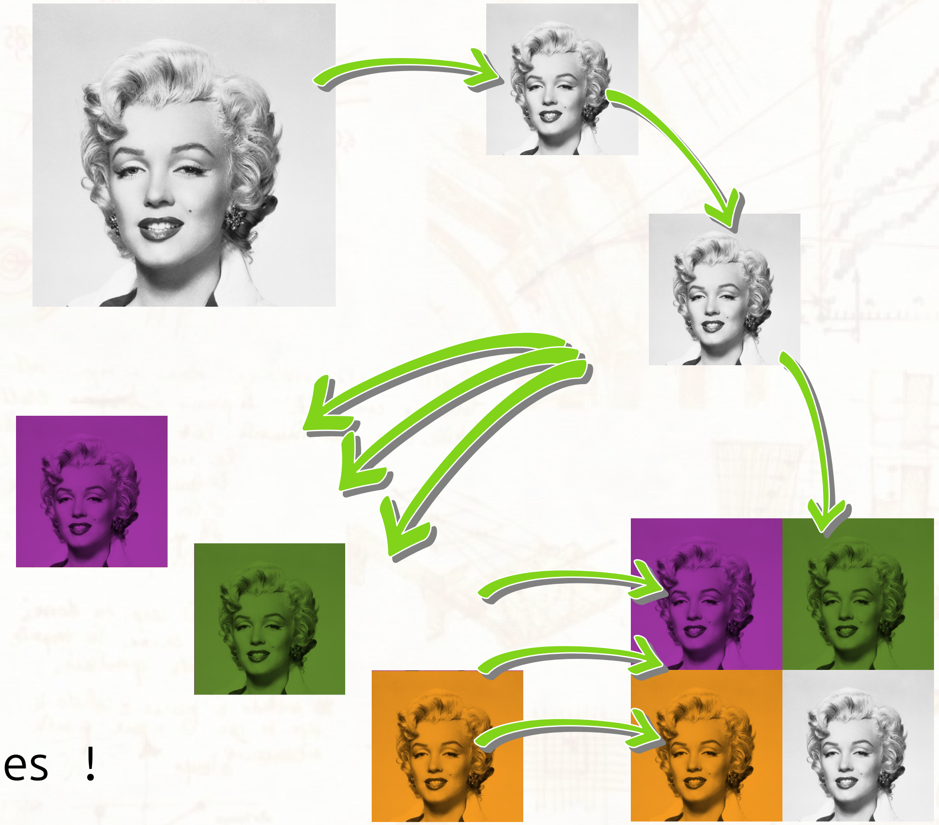
Art @ SESAME !

Art @ SESAME !

our task list

- `hands-on/parallel/20_images/test.cc`:
    - read one image from a file
    - display the image on the terminal
    - make a 0.5×0.5 smaller copy of the image
    - convert the image to gray scale
    - make tinted copies
    - combine the gray scale and tinted images into a single image with the same size as the original
    - display the image on the terminal
    - write the image to a file

- we need some libraries for handling the images !



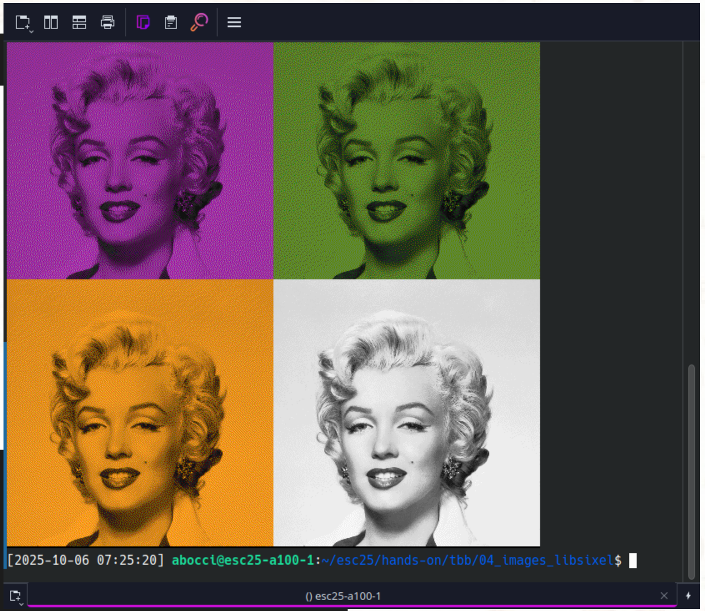
simple graphics libraries

- stb_image.h and stb_image_write.h reading and writing image files
  - header-only libraries, easy to set up

```
git clone https://github.com/nothings/stb.git
```

- libsixel
  - provides encoder/decoder implementation for DEC SIXEL graphics, and some converter programs
  - lets you display graphical images directly on the terminal
    - if your terminal supports it
  - needs to be build and installed

```
git clone git@github.com:saitoha/libsixel.git build/libsixel
cd build/libsixel
./configure --without-libcurl --without-jpeg --without-png \
    --without-pkgconfigdir --without-bashcompletiondir \
    --without-zshcompletiondir --disable-python \
    --prefix=$(realpath ../../libsixel)
make -j8 install
cd ../../
rm -rf build
```



```
[2025-10-06 07:25:20] abocci@esc25-a100-1:~/esc25/hands-on/tbb/04_images_libsixel$ 
```

cached dependencies

- if cloning and building the dependencies is slow, you can link them from a central location
- inside the exercise directory (for example `…/esc25/hands-on/tbb/04_images/`), run

```
ln -s /home/ESC/abocci/external/* .
```

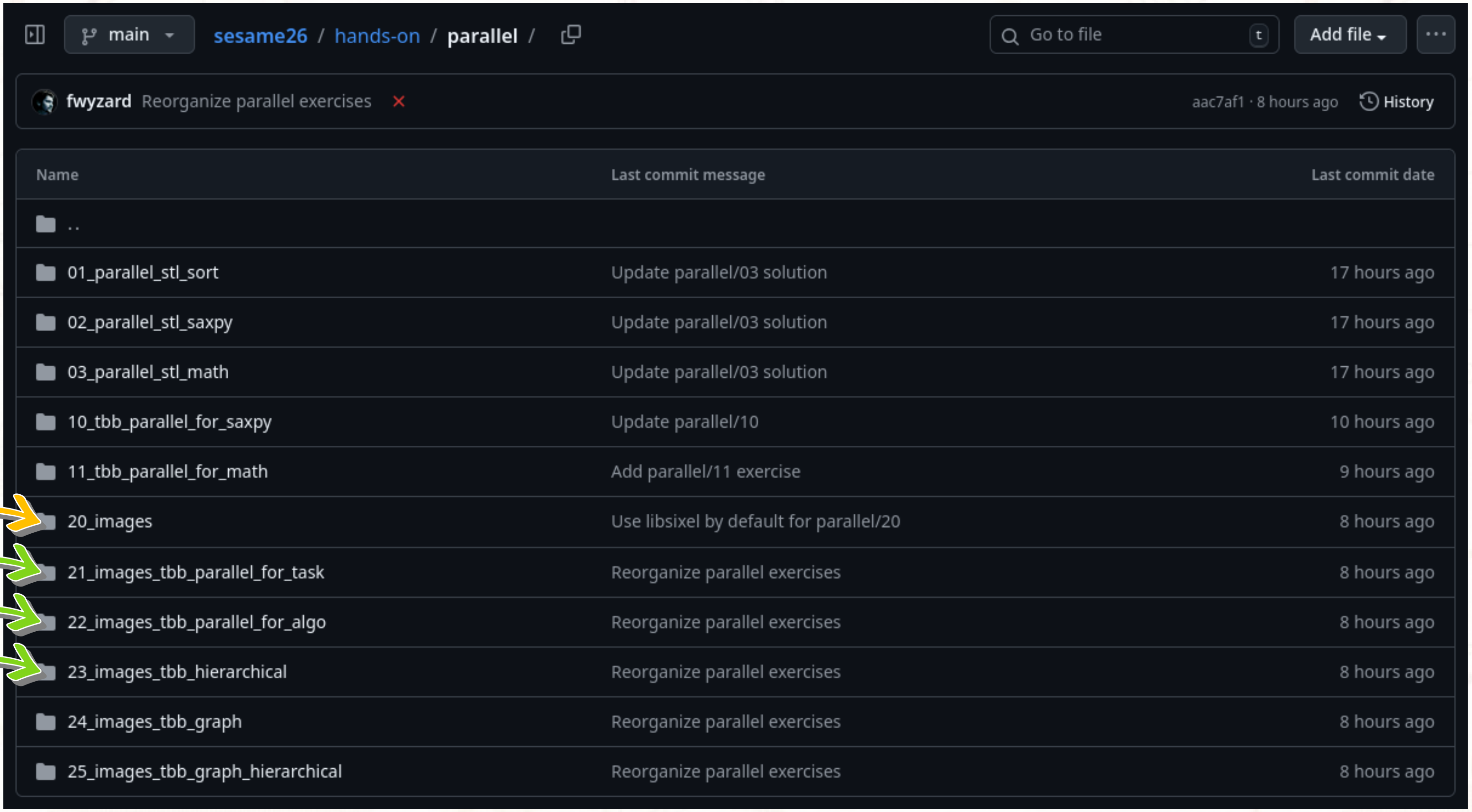
multiple levels of parallelism

- with TBB we can express multiple levels of parallelism
  - **algorithmic parallelism**: parallelise the inner loops in the various algorithms
    - scaling
    - gray scaling, tinting
    - copying
    - **very dependent on the algorithms**
  - **task-based parallelism**: parallelise the different tasks working on the same data
    - apply the different tints can be done in parallel
    - writing to disk in parallel to displaying on the terminal
    - **very dependent on the workflow**
  - **data parallelism:** process multiple images in parallel
    - weak scaling
    - **often the most efficient approach for large datasets**

- **composability**: you can also apply all of them to the same problem !

hands-on exercises

hands-on/parallel/



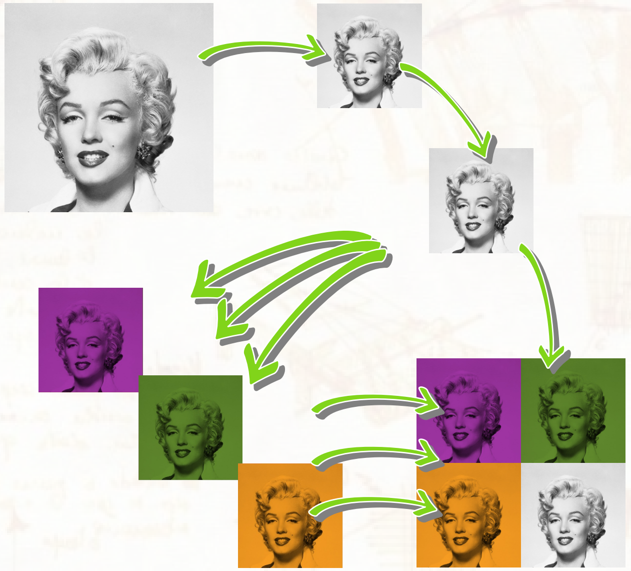
main · sesame26 / hands-on / parallel / · Go to file · Add file

fwyzard Reorganize parallel exercises ✗ aac7af1 · 8 hours ago History

Name	Last commit message	Last commit date
..		
01_parallel_stl_sort	Update parallel/03 solution	17 hours ago
02_parallel_stl_saxpy	Update parallel/03 solution	17 hours ago
03_parallel_stl_math	Update parallel/03 solution	17 hours ago
10_tbb_parallel_for_saxpy	Update parallel/10	10 hours ago
11_tbb_parallel_for_math	Add parallel/11 exercise	9 hours ago
20_images	Use libsixel by default for parallel/20	8 hours ago
21_images_tbb_parallel_for_task	Reorganize parallel exercises	8 hours ago
22_images_tbb_parallel_for_algo	Reorganize parallel exercises	8 hours ago
23_images_tbb_hierarchical	Reorganize parallel exercises	8 hours ago
24_images_tbb_graph	Reorganize parallel exercises	8 hours ago
25_images_tbb_graph_hierarchical	Reorganize parallel exercises	8 hours ago

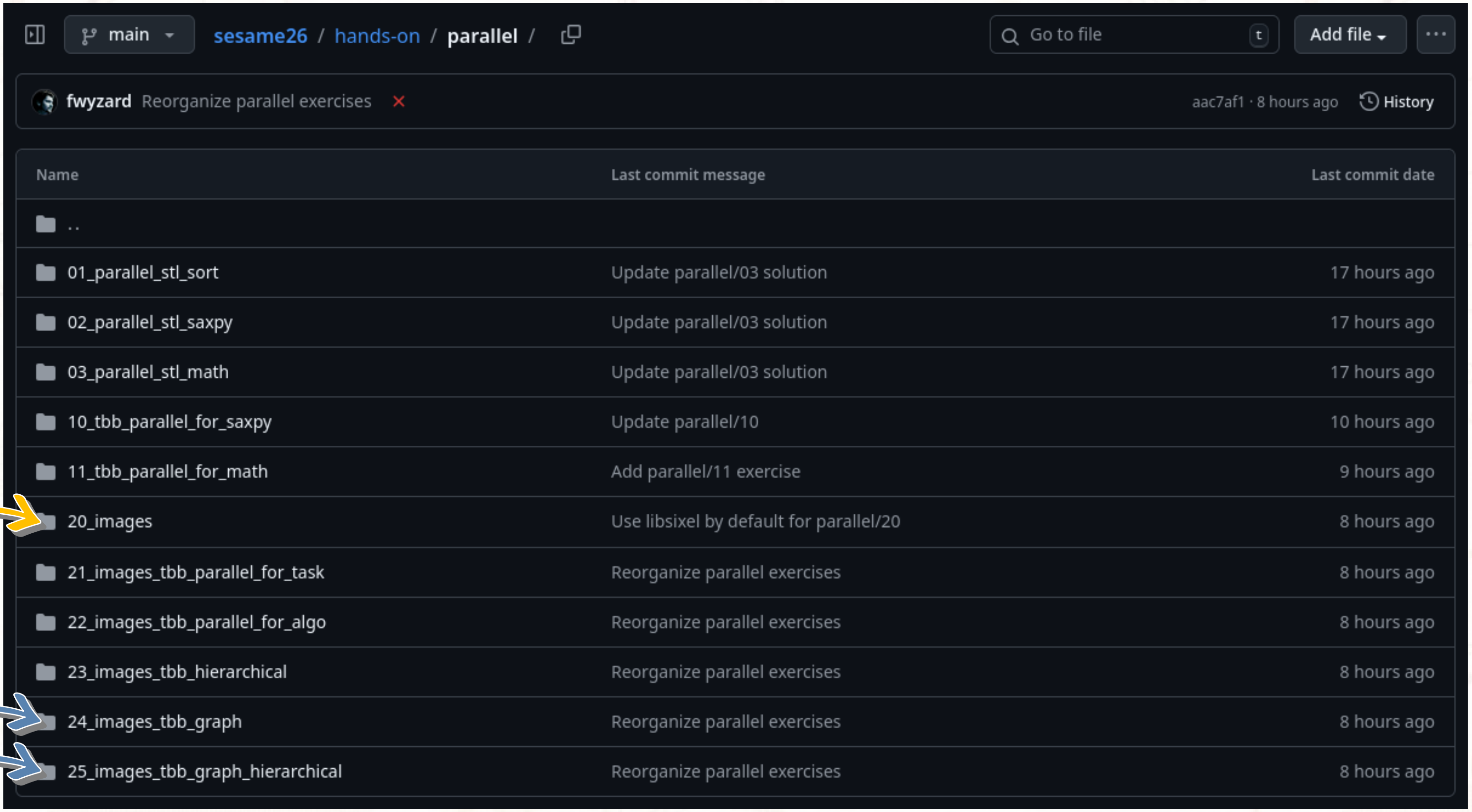
tbb::flow::graph

- `tbb::flow::graph`
  - split your program in tasks
  - declare the dependencies among tasks and expose possible parallelism
    - start from a “source” task
    - each task may depend on other tasks, forming a **directed acyclic graph** (DAG)
  - let TBB take care of scheduling the tasks
  - in our example, the tasks could be
    - loading an image
    - scaling an image
    - converting an image to greyscale
    - applying a colour tint
    - merging multiple images
    - displaying an image
    - saving an image to disk



hands-on exercises

hands-on/parallel/



main | sesame26 / hands-on / parallel /

Go to file | Add file

fwyzard Reorganize parallel exercises — aac7af1 · 8 hours ago — History

Name	Last commit message	Last commit date
..		
01_parallel_stl_sort	Update parallel/03 solution	17 hours ago
02_parallel_stl_saxpy	Update parallel/03 solution	17 hours ago
03_parallel_stl_math	Update parallel/03 solution	17 hours ago
10_tbb_parallel_for_saxpy	Update parallel/10	10 hours ago
11_tbb_parallel_for_math	Add parallel/11 exercise	9 hours ago
20_images	Use libsixel by default for parallel/20	8 hours ago
21_images_tbb_parallel_for_task	Reorganize parallel exercises	8 hours ago
22_images_tbb_parallel_for_algo	Reorganize parallel exercises	8 hours ago
23_images_tbb_hierarchical	Reorganize parallel exercises	8 hours ago
24_images_tbb_graph	Reorganize parallel exercises	8 hours ago
25_images_tbb_graph_hierarchical	Reorganize parallel exercises	8 hours ago

questions ?