35 Lezione

martedì 28 maggio 2019 14:21

PPL: parallel pattern library C# C++

TPL: task parallel library C#

Provides:

- vectorization tasks concurrent mechanism
- parallel algorithm parallel container
- agent data flow
  ↳ define block as in the data flow graph.

Vectorizing the code:

```
# pragma loop ( hint parallel (n°w))
    for ( ... ) {      → questo for loop deve essere svolto in parallelo
    }

# pragma loop (no vector)
    for ( - - ) {      → questo non viene parallelizzato
    }
```

Task

- create call allows to create a task
- Start the task
- Wait for a task
- Waitall: wait for all the task
- Continuation: you can create a task and you can put a then with the second task with t_2 that depens on t_1:

  t_1 create. then (t_2)

- there is another construct to express some other things:
  - when_all (t_8)
  - when_only (t_3)

14:43

Example:

```
x{create_task ([] {___}
  create_tasks };
auto result = when_all ( x.begin ()
                          x.end ()
```

you can build any collection of task and then ... 14:44

Parallel algorithm

- Parallel for: parallelize a for loop, whatever is the meaning of the for. Execute a number of iterations in parallel.
- Parallel for each: you have a collection (vector...) and on all the items you apply a computation.

All of them have a parameter that could be a kind of partitioner:

partition
- Static: you have a range and a number of thread. You do: $\frac{\text{\# dimension of what you are doing}}{\text{\# threads}}$ and assign this as a work for each thread.

affinity partion: Consider also some optimization.

Simple partitioner: you have a chunk size and the assignment are made in chunk

auto: It starts with partitioning and if you understand that some threads finish before the other you can assign to this threads some works taken from the others threads.

Some patterns:

Parallel_invoke

↓ starts the computation. Start 2 things in parallel, I invoke first and second element and when they finish the invoke takes back the control

```
parallel_invoke ({ [α] () {cmd1} };
                 { [γ] () {cmd2} },
                 ⋮
                 {    });
```

then return where the invoke was called

Parallel_transform

Is the equivalent of our map. takes a collection and the function that we want to apply.
3 different variant of this:
Given a collection v:

```
transform (v.begin (),
           v.end (),
           [] (int x) {return (x+x)}
```

I take the collection and for each element I obtain the new value for the specific position.

① $\forall i: x_i = f(x_i)$ oppure posso fare

② transform (v.begin(), v.end(), y.begin(), f)

$\forall i \quad y_i = f(x_i)$

③ two source collections and one destination:

transform (begin, end, begin, begin, f(binary))

$\forall i \quad y_i = f(x_i, z_i)$

Parallel_reduce

the possibility to use a binary function to sum up a collection of data.

parallel_reduce(v.begin(), v.end(), plus)

Also in this case you can use one of the partitioner (auto, simple...)

parallel_sort ( )

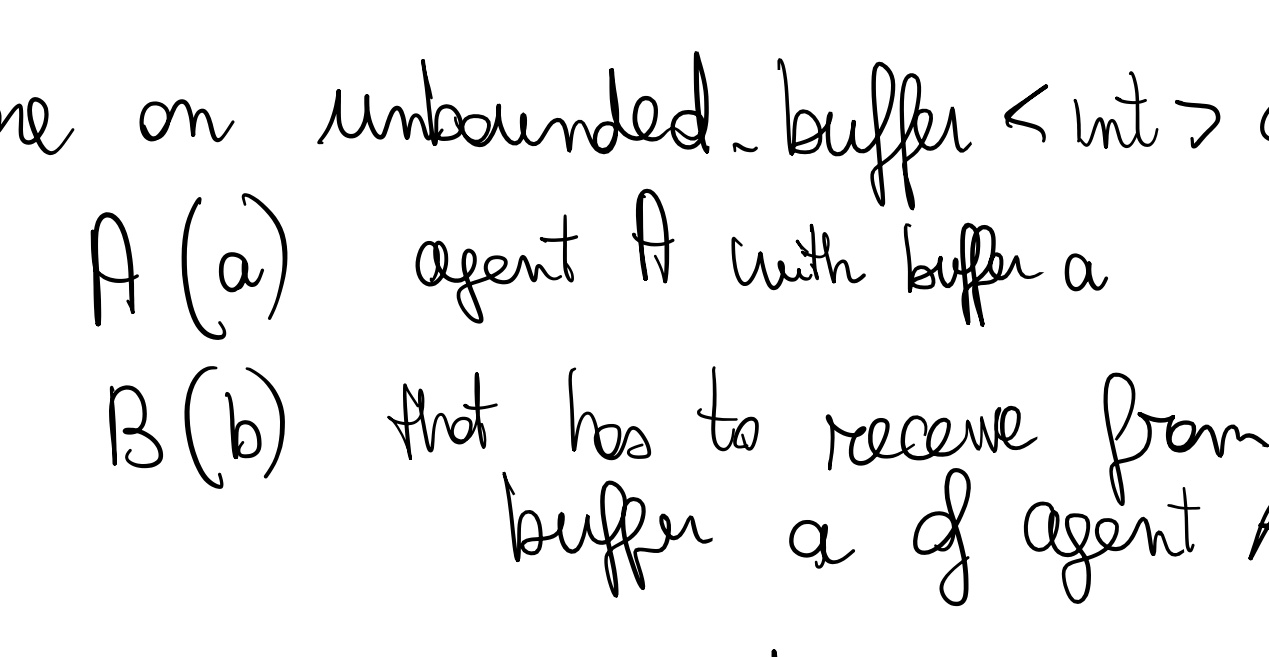
parallel_Container ( )

- Concurrent_vector: you have some operations and you want to do these operations "safe". 15:21
- Concurrent_queue: guarantees the FIFO, LIFO order of the queue/insertion and pop from the queue....

  they have:
  - Concurrent ordered/unordered map/multimap (you can have some items with the same key)
  - Concurrent ordered/unordered set/multiset

Combinable a
- local (local for the thread)
- combine method that out of all the local items get a final result.

Agents



You have some possibilities:
- Send ( )
- Receive ( )
- done ( )

B needs to know the name of A to the receive.

You can declare an unbounded_buffer <int> a
I can create A (a) agent A with buffer a
B (b) that has to receive from buffer a of agent A

Not very different from what we do with task.

Data Flow

We can declare a data flow graph using:
- buffer blocks
  - round buffer: post, receive...
  - broadcast block
  - write one blocks
- Compute blocks: is more interesting:
  - action block: can produce or consume something
  - transform
  - transform many
- group blocks: Join two thing in one single communication.

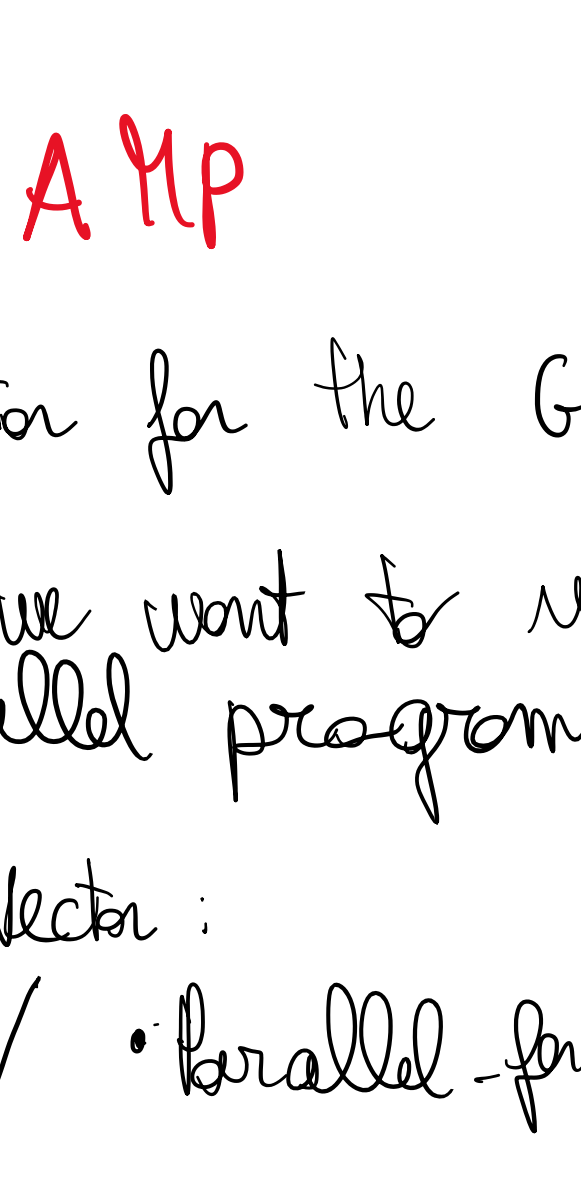
If I have a single channel batch(3) I take the first 3 values → is a way to accumulate task for consuming

I take 3 items in the first one and I output

15:42

With this you can define whatever you want.

If I have to implement a simple data flow graph:



C++ AMP

Accelerator for the GPU.

Idea: we want to use GPU to accelerate the parallel programs.

Vector:
- Parallel_for each

"View" → take a vector and create a view.

→ work in parallel → moved to accelerator } GPU