# Asynchronous IO

Hadrien Grasland

LAL – Orsay

*This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 654168.*

# Context

- Spin-off from conditions discussion, interesting on its own
- The issue in a nutshell:
  - “Regular” Algorithms are executed on the TBB thread pool
  - One thread per CPU core, if it blocks, that CPU goes idle
  - We don’t want that, so what should we do instead?
- Latest proposal: Mark algorithms as IO bound, run them on a special thread pool, keep code otherwise unchanged
- I would like to add some additional design input

# Pitfalls of (blocking) IO tasks

- Most IO resources (file descs, connections…) will lock internally or blow up when accessed from multiple threads
- Good IO APIs do not need many threads to scale
  - Modern epoll/kqueue based web servers will readily service 10~100k requests per second with a **single** IO thread
- Blocking threads come at a cost (RAM, context switches…)
  - Must keep them in small numbers and use them efficiently
- Optimal mapping of work to IO threads is resource-specific
- Piling up Scheduler complexity may bite us in the long run

# Asynchronous IO

- Many high-performance IO APIs are asynchronous, decoupling the actions of **starting IO** and **handling resuts**
- Implementations vary in complexity and scalability:
  - Blocking API wrapper: Queue requests on a blocking thread, get notified via some thread synchronization mechanism
  - Completion based: Run a callback once IO results are ready
  - Readiness based: Event loop blocking on *multiple* requests
- Growing consensus around standardized interfaces:
  - Future: Represents an individual ongoing IO operation
  - Stream: Optimization for repeated IO operations

# Strawman Gaudi integration

- IO resources are global state, modeled via Services
- These expose IO actions as asynchronous methods
  - Simplest design: individual IO method returns a future
  - Streaming design worth exploring for repetitive IO work
- IO threads become a Service implementation detail
  - Any concrete IO style may be used under the hood
  - Multiple IO Services may or not share a pool of IO threads
- Processing work can be scheduled on IO completion
  - Modern task schedulers like HPX support this natively

# Summary

- Asynchronous IO APIs are superior to blocking IO tasks
- Asynchronous IO interacts with computation scheduling
  - So leveraging it requires explicit framework support
- Synchronous APIs can be quite efficiently hidden behind an asynchronous interface, whereas the opposite is false
- Like concurrent event processing, this is the kind of basic interface concept that is best introduced early on