P3564R0: Make the concurrent forward progress guarantee usable in bulk

Mark Hoemmen, Bryce Adelstein Lelbach, and Michael Garland (NVIDIA)

SG1, WG21

Hagenberg, Austria, 2025-02

These slides were published as P3632R1.

Scheduler-generic std::execution code can never assume concurrent forward progress in bulk's function invocations, even if the scheduler's get_forward_progress_guarante

“Make bulk make progress”

- Concurrent forward progress is currently useless for bulk
  - Forward progress guarantee relates to distinct execution agents
  - Nothing specifies when different f(k) invocations run on distinct agents
  - ➔ Parallel is the strongest guarantee that generic code can assume
- Fix is mostly wording
  - Specify that each of bulk’s f(k) run on distinct execution agents
    - “Agent” is a legal fiction that need not correspond to “thread”
  - Make it ill-formed to use default (sequential) bulk with scheduler that promises concurrent forward progress
  - Let bulk report error via error channel if it can’t fulfill promise for given number of agents

Original intent: 1 execution agent per iteration

- SG1, Wrocław 2024 review of P3481R0, unanimous consent
  - “We need a version of bulk that creates an execution agent per iteration”
- P2300R0
  - Later relaxed bulk wording, only to permit default (sequential) bulk
- P2181R1 (“Correcting the design of bulk execution”)
  - Clarifying P0443R14 bulk’s forward progress wording

Why expose concurrent forward progress?

- Concurrent is effortful to implement; don’t throw it away
  - Bulk users more likely to write algorithms with explicit synchronization
  - Concurrent makes e.g., sort more efficient (e.g., fewer bulk launches)
- Parallel programming models expose it as much as possible
  - OpenMP: Concurrent across different “progress units”
  - CUDA
    - Threads in a block: can wait on each other
    - Blocks: concurrent not default, but can ask for it
  - HPX: Can ask for a “parallel section” that permits blocking sync

Run on N agents != parallel algorithm

- Parallel algorithm (e.g., for_each) has 2 parts
  - Make available N distinct execution agents
  - Distribute work items (loop iterations) to agents
- Programming models separate them
  - OpenMP: “#pragma omp parallel” vs. loop & distribution directives
  - CUDA kernel launch vs. parallel algorithms (Thrust, stdpar)
  - ScaLAPACK, High Performance Fortran (HPF)
    - Process grid / PROCESSORS directive, vs.
    - Data distributions / DISTRIBUTE (et al.) directives
- Programming models let users know & control distribution

Current wording not useful for implementing performant generic parallel algorithms

- Run on ??? number of agents
- No control over number of agents
  - Does large N oversubscribe hardware?
  - Do I need to distribute work items?

Let bulk fail if concurrent + N too large

- Implementations may not be able to promise concurrent forward progress for all possible N
- Already true of popular programming models, e.g., OpenMP
- Implementations can handle this in 2 ways
  - If recoverable: Error channel (CHANGE)
  - If not: terminate() (already permitted)