Parallelism Developments in ISO C and C++ and How to Leverage Them

Bryce Adelstein Lelbach <brycelelbach@gmail.com>
@blelbach
github.com/brycelelbach
std::vector<T> x = // ...
#pragma omp parallel for simd
for (std::size_t i = 0; i < x.size(); ++i)
    process(x[i]);

std::vector<T> x = // ...
std::for_each(std::par_unseq,
              x.begin(), x.end(), process);
C++ Executor Model

<table>
<thead>
<tr>
<th>Property</th>
<th>C++ Concept Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution restrictions</td>
<td><code>ExecutionPolicy</code> (in Parallelism TS v1 and C++17)</td>
</tr>
<tr>
<td>Sequence of execution</td>
<td><code>Executor</code> (targeted for Parallelism TS v2 and C++20)</td>
</tr>
<tr>
<td>Where execution happens</td>
<td><code>Executor</code> (targeted for Parallelism TS v2 and C++20)</td>
</tr>
<tr>
<td>Grain size of work items</td>
<td><code>ExecutorParameter</code></td>
</tr>
</tbody>
</table>

- Asynchronous task creation:
  - `async(ExecutorPolicy&&, ...)`

- Parallel algorithms:
  - `for_each(ExecutorPolicy&&, ...), sort(ExecutorPolicy&&, ...)`
thrust::gpu_executor gpu = // ...
std::vector<T, thrust::pmr::allocator> x = // ...

std::sort(std::par_unseq.on(gpu),
    x.begin(), x.end());

my::thread_pool_executor tp = // ...
std::vector<T> x = // ...

std::sort(std::par_unseq.on(tp),
    x.begin(), x.end());
std::vector<double> x = // ...
std::vector<double> y = // ...

double d = 0.0;

int i;

#pragma omp parallel shared(x, y) private(i)
#pragma omp for reduction(+ : d)
for (i = 0; i < x.size(); ++i)
    d = d + x[i] * y[i];
std::vector<double> x = // ...
std::vector<double> y = // ...

double d =
    std::transform_reduce(std::execution::par_unseq,
      x.begin(), x.end(), y.begin());
bool is_word_beginning(char left, char right) {
    return std::isspace(left) && !std::isspace(right);
}

std::size_t word_count(std::string_view s) {
    if (s.empty()) return 0;

    std::size_t wc =
        std::transform_reduce(
            std::execution::par_unseq,
            s.begin(), s.end() - 1,
            s.begin() + 1,
            std::size_t(!std::isspace(s.front()) ? 1 : 0),
            std::plus<std::size_t>(),
            is_word_beginning
        );

    return wc;
}
• Non-HPC domains are starting to have the same problems and needs that we have; this is good!
  • ~7 million C users worldwide (source: JetBrains).
  • ~5 million C++ users worldwide (source: JetBrains).

• There is growing interest in standardizing parallel programming features in C and C++.
  • C++17 parallel algorithms.
  • Proposed C++20 executors.
  • CPLEX study group for C parallelism extensions.
  • Future efforts to extend the C and C++ memory models for RMA/shmem and heterogeneous memory.

• The C and C++ standards get significant adoption with both users and vendors.
  • C and C++ have driven innovations in compiler optimization and will continue to do so.
  • OpenMP is cute, but optional. C and C++ features must be implemented and quality of implementation must be high or some of our ~12 million users will be unhappy.
• Chapel can leverage the machinery for C++17/C++20/C++23/C2x parallelism.

• Clang/LLVM Coroutines.
  • Code transform facilities that facilitate asynchronous programming and a powerful set of LLVM optimizations for them.
  • Based on the C++ Coroutines TS.
  • Available in Clang/LLVM trunk as of this week.

• LLVM Parallel Intermediate Representation.
  • Extensions to the LLVM IR that express parallelism constructs as first class entities and facilitate the development of LLVM parallelism optimization passes.

Copyright 2017, Bryce Adelstein Lelbach
• Chapel can leverage the machinery for C++17/C++20/C++23/C2x parallelism.

• libc++ Parallel Runtime Interface
  • Low-level backend interface used by the libc++ implementation of the C++17 parallel algorithms, designed to be targetable by 3rd-party runtimes that want to interoperate with C++17 parallel algorithms.
  • Longer-term goal: one unified parallel runtime interface for LLVM, covering C++17 parallel algorithms, OpenMP, OpenACC, OpenCL, etc.
  • Design work is starting in the near future; this is a great time to get involved.
• Chapel can leverage the machinery for C++17/C++20/C++23/C2x parallelism.

• LLVM Polly
  • High-level loop and data-locality optimizer and optimization infrastructure for LLVM.
  • “Early” loop pass, similar to the Intel compiler’s vectorizer and unlike LLVM’s production vectorizer (which is a “late” vectorizer).
  • Performs traditional loop optimizations, e.g. tiling and loop fusion, as well as nested loop optimizations, e.g. loop order interchange.
  • Can also exploit OpenMP-level parallelism and expose vectorization opportunities to the backend.
• Chapel can leverage the machinery for C++17/C++20/C++23/C2x parallelism.

• Machine-Learning Compiler Optimizations
  • Possible uses: making cost-modelling decisions such as auto-vectorization and auto-parallelization.
  • HPX team has developed Clang/LLVM-based machine-learning compiler techniques (deciding to parallelize or not, what chunk size to use) for the reference implementation of the C++17 parallel algorithms.
  • Look for Zahra Khatami’s papers and posters – SC16, IPDPS17, PDSEC17.
Good time to get involved in C and C++!

Bryce Adelstein Lelbach <brycelelbach@gmail.com>
@blelbach
github.com/brycelelbach