Compiling in Parallel

Tools and Libraries for Compile-time Software Engineering

HiPEAC Conference 2024
Munich, Germany

Paul Keir [1]
Joel FALCOU [2]

[1]School of Computing, Engineering & Physical Sciences
University of the West of Scotland, Paisley, UK

[2]Le Laboratoire Interdisciplinaire des Sciences du Numérique (LISN)
Université Paris-Saclay, Paris, France

January 17th, 2024

Overview

- C++ Extensions for Parallelism
- Parallel Evaluation within the ClangOz Compiler
- A New Execution Policy Extension for the Standard Library
- Benchmark Program Analysis
- Related Work
- Conclusion and Future Work

C++ Extensions for Parallelism

- Introduced to the standard *algorithms* library in C++17
- A set of overloads of existing standard library algorithms
- Each function template accepts a new *execution policy* parameter

```
constexpr std::execution::sequenced_policy            seq{};
constexpr std::execution::parallel_policy             par{};
constexpr std::execution::parallel_unsequenced_policy par_unseq{};
constexpr std::execution::unsequenced_policy          unseq{}; // C++20
```

- The first `for_each` below increments all elements of `v` in serial
- The `std::execution::par` object can permit parallel execution
- Common serial execution is also available via `std::execution::seq`

```
std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
std::for_each(v.begin(), v.end(), [](int& i) { i++; });
std::for_each(std::execution::par, v.begin(), v.end(), [](int& i) { i++; });
std::for_each(std::execution::seq, v.begin(), v.end(), [](int& i) { i++; });
assert(v == (std::vector{4, 5, 6, 7, 8, 9, 10, 11}));
```

Parallel *for loops*: the ClangOz Low-Level Intrinsics API

The following conditions must be met for a *for loop* to be parallelised:

1. The targeted loop must be within a `constexpr` function;
2. The function must include an `execution::par` parameter;
3. The loop must come immediately after the ClangOz intrinsics:

Parallel *for loops*: the ClangOz Low-Level Intrinsics API

The following conditions must be met for a *for loop* to be parallelised:

1. The targeted loop must be within a `constexpr` function;
2. The function must include an `execution::par` parameter;
3. The loop must come immediately after the ClangOz intrinsics:

```
template <class T, class U>
constexpr
void __BeginEndIteratorPair(T& Begin, U& End);

template <class T, class U>
constexpr
void __PartitionUsingIndex(T LHS, U RHS, RelationalType RelTy);

template <class T>
constexpr
void __IteratorLoopStep(T& StartIter, OperatorType OpTy, const T& BoundIter);

template <class T>
constexpr
void __ReduceVariable(T Var, ReductionType RedTy, OperatorType OpTy);
```

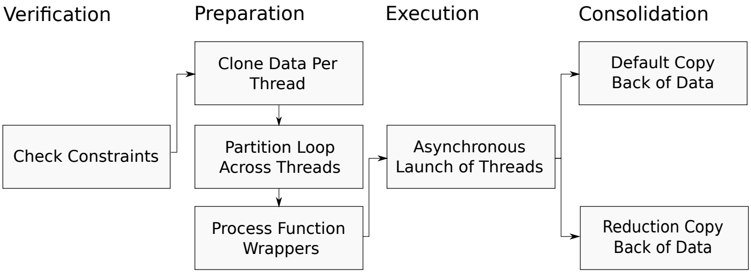
Compiler intrinsics: look like functions; no bodies; no-ops at runtime

Limitations of the ClangOz compiler

- There is no support for nested parallelism
  - ...only the outer loop of a loop nest is parallelised;
- Only one loop can be parallelised within each function;
- Only containers owning contiguous data may be used;
  - **ContiguousContainer**
    `std::string`, `std::vector`, `std::array`, and built-in arrays (`T[]`)
- Containers must be built upon pointer-based iterators.

Parallelisation Phases within ClangOz

- `CallStackFrame` acts as a call stack for the constant evaluator
- `EvalInfo` holds information about the expression being evaluated

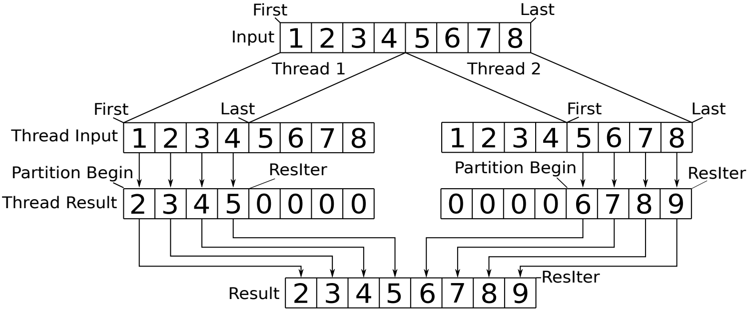


1. Check the 3 conditions are met; ensure no other active parallelism
2. `EvalInfo` clones; partition iteration space statically between threads
3. Launch, execute and await completion; last thread takes remainder
4. Synchronise thread data back into the main thread's `EvalInfo`

Parallel Decomposition via `PartitionedOrderedAssign`

```
__BeginEndIteratorPair(First, Last);
__IteratorLoopStep(ResIter, OperatorType::PreInc);
__ReduceVariable(ResIter, ReductionType::PartitionedOrderedAssign,
                 OperatorType::PreInc);

for (; First != Last; ++First) { *ResIter = (*First) + 1; ++ResIter; }
```

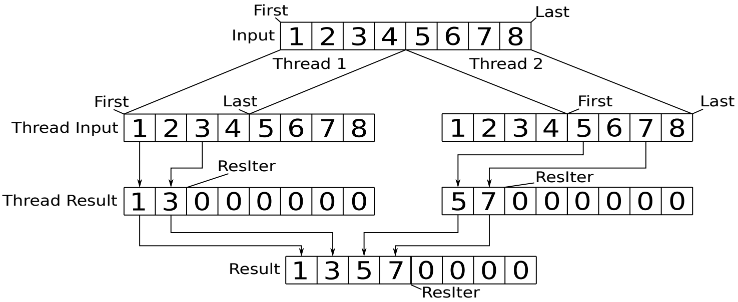


- `__IteratorLoopStep` asserts the loop/operator updates the pointer
- `PartitionedOrderedAssign` identifies a regular partitioning

Parallel Decomposition via `OrderedAssign`

```
__BeginEndIteratorPair(First, Last);
__ReduceVariable(ResIter, ReductionType::OrderedAssign,
                 OperatorType::PreInc);

for (; First!= Last; ++First)
  if (*First % 2) { *ResIter = *First; ++ResIter; }
```



- Here an irregular partitioning is employed via `OrderedAssign`
- Useful when each thread creates distinct quantities for aggregation

Implementing `for_each` using the Low-Level API

- But we don't want end users working with the low-level API
- Instead, we react to the *execution policy*: `std::execution::par`
  - ...when evaluated within a constant expression

Implementing for_each using the Low-Level API

- But we don't want end users working with the low-level API
- Instead, we react to the *execution policy*: `std::execution::par`
  - ...when evaluated within a constant expression

Below we implement the standard function template `std::for_each`:

```
template <class _ExecutionPolicy, class _ForwardIterator, class _Function>
constexpr
__enable_if_constexpr_par_execution_policy_t<_ExecutionPolicy, void>
for_each(_ExecutionPolicy&& __exec, _ForwardIterator __first,
         _ForwardIterator __last, _Function __f)
{
  __BeginEndIteratorPair(__first, __last);
  __ReduceVariable(__first, PartitionedOrderedAssign, PreInc);

  for (; __first != __last; ++__first)
    __f(*__first);
}
```

We have implemented 30 function templates from the C++ standard Algorithms and Numerics libraries

Example Usage of the High-Level API

```
constexpr bool g() {
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
  std::for_each(std::execution::par, v.begin(), v.end(),
                [](int &i) { i++; });
  assert(v == (std::vector{2, 3, 4, 5, 6, 7, 8, 9}));
}
static_assert(g() == true);
```

- The function `g` increments each element of `v` in parallel
- The `static_assert` invokes constant expression evaluation
- Other execution policy overloads are not `constexpr`
- Traditional `std::for_each` *is* `constexpr`; but serial only

Example Usage of the High-Level API

```
constexpr bool g() {
  std::vector<int> v{1, 2, 3, 4, 5, 6, 7, 8};
  std::for_each(std::execution::par, v.begin(), v.end(),
                [](int &i) { i++; });
  assert(v == (std::vector{2, 3, 4, 5, 6, 7, 8, 9}));
}
static_assert(g() == true);
```

- The function `g` increments each element of `v` in parallel
- The `static_assert` invokes constant expression evaluation
- Other execution policy overloads are not `constexpr`
- Traditional `std::for_each` *is* `constexpr`; but serial only
  - ...though C'est 2 does include some `constexpr` `std::execution::seq` overloads

Benchmarking Information

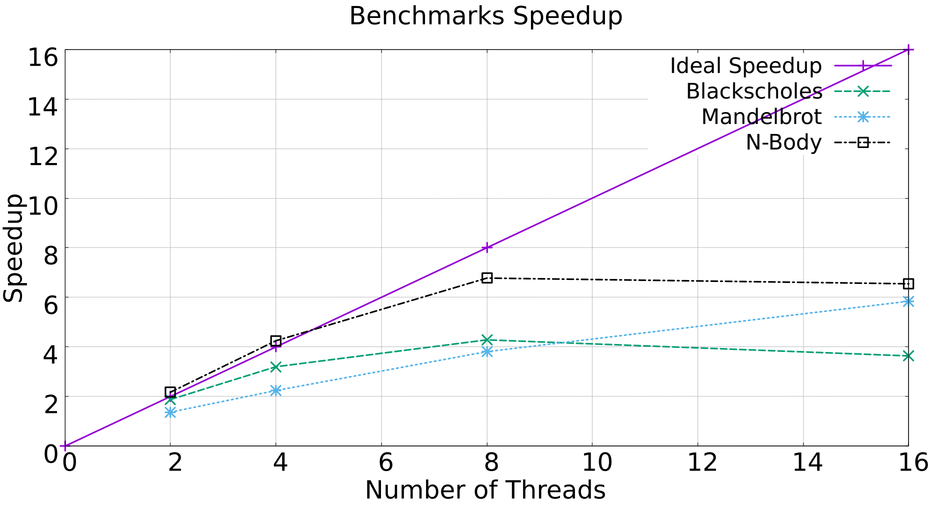
- Intel Core i9-12900K CPU; 32 GB RAM; 1 TB SSD
- 8 performance cores; 16 hyper-threads
- 64-bit Ubuntu under WSL2 virtual machine on Windows 11
- Executed using 2, 4, 8 and 16 threads; times averaged over 6 runs
- Times correspond to one or more consecutive `for_each` invocations

The Benchmark Programs:

- **Mandelbrot*** Complex value iteration; max 128 iterations
- **Blackscholes**** processes financial data using a PDE
- **N-Body*** Jovian planets simulation via a symplectic-integrator

* From The Computer Language Benchmarks Game
** From Princeton's PARSEC Benchmark Suite

Overall Speedup



Related Work

Optimizing gpu programs by partial evaluation
A. Tyurin, D. Berezun, and S. Grigorev
Proceedings of 25th ACM PPoPP (2020)

Anydsl: A partial evaluation framework for programming high-performance libraries
R. Leißa, et al.
Proceedings of ACM on Programming Languages, OOPSLA (2018)

Practical partial evaluation for high-performance dynamic language runtimes
T. Würthinger et al.
Proceedings of the 38th ACM SIGPLAN Conference on PLDI (2017)

Large-scale parallelization of partial evaluations in evolutionary algorithms for real-world problems
A. Bouter, T. Alderliesten, A. Bel, C. Witteveen, and P. A. Bosman
Proceedings of the Genetic and Evolutionary Computation Conference (2018)

Distributed partial evaluation
M. Sperber, P. Thiemann, and H. Klaeren
Proceedings of the 2nd International Symposium on Parallel Symbolic Computation (1997)

Partial evaluation in parallel
C. Consel and O. Danvy
Lisp and Symbolic Computation (1993)

Conclusion

- ClangOz, the first compiler capable of parallel evaluation of C++ constant expressions
- Support (subset) for the C++ Standard Algorithms & Numerics library
  - ...via the `std::execution::par` policy overloads
- A flexible, low-level intrinsics API
- A suite of `constexpr` benchmark programs, with study results

`https://github.com/agozillon/ClangOz`

Future Work

- Add support for more C++ standard library functions
- Profile, to improve parallel performance

Hands On

- wordcount; todo

Acknowledgements

UWS UNIVERSITY OF THE WEST of SCOTLAND

RSE The Royal Society of Edinburgh
KNOWLEDGE MADE USEFUL